UCCA Documentation

Release 1.3.4

Daniel Hershcovich

Jun 23, 2020

Contents:

UCCA is a linguistic framework for semantic annotation, whose details are available at the following paper:

```
@inproceedings{abend2013universal,
  author={Abend, Omri  and  Rappoport, Ari},
  title={{U}niversal {C}onceptual {C}ognitive {A}nnotation ({UCCA})},
  booktitle={Proc. of ACL},
  month={August},
  year={2013},
  pages={228--238},
  url={http://aclweb.org/anthology/P13-1023}
}
```

This Python 3 package provides an API to the UCCA annotation and tools to manipulate and process it. Its main features are conversion between different representations of UCCA annotations, and rich objects for all of the linguistic relations which appear in the theoretical framework (see `core`, `layer0`, `layer1` and `convert` modules under the `ucca` package).

The `scripts` package contains various utilities for processing passage files.

To parse text to UCCA graphs, use TUPA, the UCCA parser.

CHAPTER 1

Authors

- Amit Beka: amit.beka@gmail.com
- Daniel Hershcovich: daniel.hershcovich@gmail.com

CHAPTER 2

License

This package is licensed under the GPLv3 or later license.

```
[ ~ Dependencies scanned by PyUp.io ~ ]
```

For more information about how to use this library, see the *API Documentation*.

2.1 API Documentation

2.1.1 Getting Started

To load UCCA passages from XML files, manipulate them and write to files, use the following code template:

```
from ucca.ioutil import get_passages_with_progress_bar, write_passage
for passage in get_passages_with_progress_bar(filenames):
    ...
    write_passage(passage)
```

Each passage instantiates the *ucca.core.Passage* class.

XML files can be downloaded from the various UCCA corpora.

2.1.2 ucca.constructions Module

Functions

add_argument(argparser[, default])	
create_category_construction(tag)	

Continued on next page

Table 1 – continued from previous page

create_passage_yields(p, *args[, tags])	**param p** passage to find terminal yields of
diff_terminals(*passages)	
extract_candidates(passage[, constructions, ...])	Find candidate edges by constructions in UCCA passage.
get_by_name(name)	
get_by_names([names])	
positions(terminals)	
terminal_ids(passage)	
verify_terminals_match(passage, reference)	

add_argument

ucca.constructions.**add_argument**(*argparser*, *default=True*)

create_category_construction

ucca.constructions.**create_category_construction**(*tag*)

create_passage_yields

ucca.constructions.**create_passage_yields**(*p*, **args*, *tags=True*, ***kwargs*)

Parameters

- **p** – passage to find terminal yields of
- **tags** – instead of Candidates, map simply to their edge tags

Returns

dict: Construction -> dict: set of terminal indices (excluding punctuation) ->

list of Candidates whose yield (excluding remotes and punctuation) is that set

diff_terminals

ucca.constructions.**diff_terminals**(**passages*)

extract_candidates

ucca.constructions.**extract_candidates**(*passage*, *constructions=None*, *reference=None*, *reference_yield_tags=None*, *verbose=False*)

Find candidate edges by constructions in UCCA passage. :param passage: Passage object to find constructions in :param constructions: list of constructions to include or None for all :param reference: Passage object to get POS tags from, and categories for fine-grained scores (default: 'passage') :param reference_yield_tags: yield tags from reference passage for fine-grained evaluation:

dict: set of terminal indices (excluding punctuation) -> list of edges of the Construction whose yield (excluding remotes and punctuation) is that set

Parameters **verbose** – whether to print tagged text

Returns dict of Construction -> list of corresponding Candidates

get_by_name

ucca.constructions.**get_by_name**(*name*)

get_by_names

ucca.constructions.**get_by_names**(*names=None*)

positions

ucca.constructions.**positions**(*terminals*)

terminal_ids

ucca.constructions.**terminal_ids**(*passage*)

verify_terminals_match

ucca.constructions.**verify_terminals_match**(*passage*, *reference*)

Classes

Candidate(edge[, reference, ...])	
Categories()	
Construction(name, description, criterion[, ...])	
EdgeTags	Layer 1 Edge tags.
NodeTags	Layer 1 Node tags.
OrderedDict	Dictionary that remembers insertion order
chain	chain(*iterables) –> chain object

Candidate

class ucca.constructions.**Candidate**(*edge*, *reference=None*, *reference_yield_tags=None*, *verbose=False*)

Bases: object

Attributes Summary

dep
excluded
heads

Continued on next page

Table 3 – continued from previous page

implicit
pos
remote
tokens

Methods Summary

constructions([constructions])
is_implicit()
is_predicate()
is_primary()
is_punct()
is_remote()
terminal_yield(construction)

Attributes Documentation

dep

excluded

heads

implicit

pos

remote

tokens

Methods Documentation

constructions(*constructions=None*)

is_implicit()

is_predicate()

is_primary()

is_punct()

is_remote()

terminal_yield(*construction*)

Categories

class ucca.constructions.**Categories**

Bases: *ucca.constructions.Construction*

Methods Summary

__call__(candidate)	Call self as a function.

Methods Documentation

__call__ (*candidate*)
Call self as a function.

Construction

class ucca.constructions.**Construction**(*name*, *description*, *criterion*, *default=False*)
Bases: object

Attributes Summary

is_punct

Methods Summary

__call__(candidate)	Call self as a function.

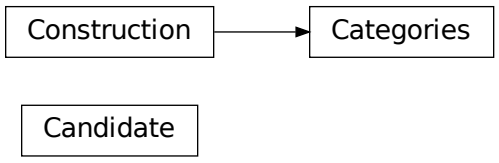
Attributes Documentation

is_punct

Methods Documentation

__call__ (*candidate*)
Call self as a function.

Class Inheritance Diagram



2.1.3 ucca.convert Module

Converter module between different UCCA annotation formats.

This module contains utilities to convert between UCCA annotation in different forms, to/from the `core`.Passage form, acts as a pivot for all conversions.

The possible other formats are: site XML standard XML conll (CoNLL-X dependency parsing shared task) sdp (SemEval 2015 semantic dependency parsing shared task)

Functions

`attach_punct`(l0, l1)	
file2passage(filename)	Opens a file and returns its parsed Passage object Tries to read both as a standard XML file and as a binary pickle :param filename: file name to write to
from_json(lines, *args[, ...])	Convert text (or dict) in UCCA-App JSON format to a Passage object.
from_site(elem)	Converts site XML structure to `core`.Passage object.
from_standard(root[, extra_funcs])	
from_text(text[, passage_id, tokenized, ...])	Converts from tokenized strings to a Passage object.
get_categories_details(d)	
get_json_attrib(d)	
join_passages(passages[, passage_id, remarks])	Join passages to one passage with all the nodes in order :param passages: sequence of passages to join :param passage_id: ID of newly created passage (otherwise, ID of first passage) :param remarks: add original node ID as remarks to the new nodes :return: joined passage
passage2file(passage, filename[, indent, binary])	Writes a UCCA passage as a standard XML file or a binary pickle :param passage: passage object to write :param filename: file name to write to :param indent: whether to indent each line :param binary: whether to write pickle format (or XML)
pickle2passage(filename)	
split2paragraphs(passage[, remarks, lang, ids])	
split2segments(passage, is_sentences[, ...])	Split passage to sub-passages :param passage: Passage object :param is_sentences: if True, split to sentences; otherwise, paragraphs :param remarks: Whether to add remarks with original node IDs :param lang: language to use for sentence splitting model :param ids: optional iterable of ids to set passage IDs for each split :return: sequence of passages
split2sentences(passage[, remarks, lang, ids])	
split_passage(passage, ends[, remarks, ids, ...])	Split the passage on the given terminal positions :param passage: passage to split :param ends: sequence of positions at which the split passages will end :param remarks: add original node ID as remarks to the new nodes :param ids: optional iterable of ids, the same length as ends, to set passage IDs for each split :param suffix_format: in case ids is None, use this format for the running index suffix :param suffix_start: in case ids is None, use this starting index for the running index suffix :return: sequence of passages

Continued on next page

Table 8 – continued from previous page

to_json(passage, *args[, return_dict, ...])	Convert a Passage object to text (or dict) in UCCA-App JSON :param passage: the Passage object to convert :param return_dict: whether to return dict rather than list of lines :param tok_task: either None (to do tokenization too), or a completed tokenization task dict with token IDs, or True, to indicate that the function should do only tokenization and not annotation :param all_categories: list of category dicts so that IDs can be added, if available - otherwise names are used :param skip_category_mapping: if False, translate edge tag abbreviations to category names; if True, don't :return: list of lines in JSON format if return_dict=False, or task dict if True
to_sequence(passage)	Converts from a Passage object to linearized text sequence.
to_site(passage)	Converts a passage to the site XML format.
to_standard(passage)	Converts a Passage object to a standard XML root element.
to_text(passage[, sentences, lang])	Converts from a Passage object to tokenized strings.
xml2passage(filename)	

file2passage

ucca.convert.**file2passage**(*filename*)

Opens a file and returns its parsed Passage object Tries to read both as a standard XML file and as a binary pickle :param filename: file name to write to

from_json

ucca.convert.**from_json**(*lines*, **args*, *skip_category_mapping=False*, *by_external_id=False*, ***kwargs*)

Convert text (or dict) in UCCA-App JSON format to a Passage object.

According to the API, annotation units are organized in a tree, where the full unit is included as a child of its parent: https://github.com/omriabnd/UCCA-App/blob/master/UCCAApp_REST_API_Reference.pdf Token children are included in full in the "children_tokens" field. Note: children_tokens contains all tokens that are descendants of the unit, not just immediate children.

tree_id: encodes the path leading to the node, e.g., 3-5-2. 1-based, and in reverse order to the children's appearance, so that 1 is last, 2 is before last, etc. The exception is the first level, where there is just 0, and the next level starts from 1 (not 0-1).

parent_tree_id: the tree_id of the node's parent, where 0 is the root

Parameters

- **lines** – iterable of lines in JSON format, describing a single passage.
- **skip_category_mapping** – if False, translate category names to edge tag abbreviations; if True, don't
- **by_external_id** – set passage ID to be the external ID of the source passage rather than its ID

Returns generator of Passage objects

from_site

ucca.convert.**from_site**(*elem*)

Converts site XML structure to core.Passage object.

Parameters **elem** – root element of the XML structure

Returns The converted core.Passage object

from_standard

ucca.convert.**from_standard**(*root*, *extra_funcs=None*)

from_text

ucca.convert.**from_text**(*text*, *passage_id='1'*, *tokenized=False*, *one_per_line=False*, *extra_format=None*, *lang='en'*, *return_text=False*, **args*, ***kwargs*)

Converts from tokenized strings to a Passage object.

Parameters

- **text** – a multi-line string or a sequence of strings: each line will be a new paragraph, and blank lines separate passages
- **passage_id** – prefix of ID to set for returned passages
- **tokenized** – whether the text is already given as a list of tokens
- **one_per_line** – each line will be a new passage rather than just a new paragraph
- **extra_format** – value to set in passage.extra["format"]
- **lang** – language to use for tokenization model
- **return_text** – whether to return the original text with each passage and not just the passage itself

Returns generator of Passage object with only Terminal units

get_categories_details

ucca.convert.**get_categories_details**(*d*)

get_json_attrib

ucca.convert.**get_json_attrib**(*d*)

join_passages

ucca.convert.**join_passages**(*passages*, *passage_id=None*, *remarks=False*)

Join passages to one passage with all the nodes in order :param passages: sequence of passages to join :param passage_id: ID of newly created passage (otherwise, ID of first passage) :param remarks: add original node ID as remarks to the new nodes :return: joined passage

passage2file

ucca.convert.**passage2file**(*passage*, *filename*, *indent=True*, *binary=False*)

Writes a UCCA passage as a standard XML file or a binary pickle :param passage: passage object to write :param filename: file name to write to :param indent: whether to indent each line :param binary: whether to write pickle format (or XML)

pickle2passage

ucca.convert.**pickle2passage**(*filename*)

split2paragraphs

ucca.convert.**split2paragraphs**(*passage*, *remarks=False*, *lang='en'*, *ids=None*)

split2segments

ucca.convert.**split2segments**(*passage*, *is_sentences*, *remarks=False*, *lang='en'*, *ids=None*)

Split passage to sub-passages :param passage: Passage object :param is_sentences: if True, split to sentences; otherwise, paragraphs :param remarks: Whether to add remarks with original node IDs :param lang: language to use for sentence splitting model :param ids: optional iterable of ids to set passage IDs for each split :return: sequence of passages

split2sentences

ucca.convert.**split2sentences**(*passage*, *remarks=False*, *lang='en'*, *ids=None*)

split_passage

ucca.convert.**split_passage**(*passage*, *ends*, *remarks=False*, *ids=None*, *suffix_format='%03d'*, *suffix_start=0*)

Split the passage on the given terminal positions :param passage: passage to split :param ends: sequence of positions at which the split passages will end :param remarks: add original node ID as remarks to the new nodes :param ids: optional iterable of ids, the same length as ends, to set passage IDs for each split :param suffix_format: in case ids is None, use this format for the running index suffix :param suffix_start: in case ids is None, use this starting index for the running index suffix :return: sequence of passages

to_json

ucca.convert.**to_json**(*passage*, **args*, *return_dict=False*, *tok_task=None*, *all_categories=None*, *skip_category_mapping=False*, ***kwargs*)

Convert a Passage object to text (or dict) in UCCA-App JSON :param passage: the Passage object to convert :param return_dict: whether to return dict rather than list of lines :param tok_task: either None (to do tokenization too), or a completed tokenization task dict with token IDs,

or True, to indicate that the function should do only tokenization and not annotation

Parameters

- **all_categories** – list of category dicts so that IDs can be added, if available - otherwise names are used
- **skip_category_mapping** – if False, translate edge tag abbreviations to category names; if True, don't

Returns list of lines in JSON format if return_dict=False, or task dict if True

to_sequence

ucca.convert.**to_sequence**(*passage*)

Converts from a Passage object to linearized text sequence.

Parameters **passage** – the Passage object to convert

Returns a list of strings - 1 if sentences=False, # of sentences otherwise

to_site

ucca.convert.**to_site**(*passage*)

Converts a passage to the site XML format.

Parameters **passage** – the passage to convert

Returns the root element of the standard XML structure

to_standard

ucca.convert.**to_standard**(*passage*)

Converts a Passage object to a standard XML root element.

The standard XML specification is not contained here, but it uses a very shallow structure with attributes to create hierarchy.

Parameters **passage** – the passage to convert

Returns the root element of the standard XML structure

to_text

ucca.convert.**to_text**(*passage*, *sentences=True*, *lang='en'*, **args*, ***kwargs*)

Converts from a Passage object to tokenized strings.

Parameters

- **passage** – the Passage object to convert
- **sentences** – whether to break the Passage to sentences (one for string) or leave as one string. Defaults to True
- **lang** – language to use for sentence splitting model

Returns a list of strings - 1 if sentences=False, # of sentences otherwise

xml2passage

ucca.convert.**xml2passage**(*filename*)

Classes

EdgeTags	Layer 1 Edge tags.
JSONDecodeError(msg, doc, pos)	Subclass of ValueError with the following additional properties:
SiteCfg	Contains static configuration for conversion to/from the site XML.
SiteUtil	Contains utility functions for converting to/from the site XML.
SiteXMLUnknownElement	
attrgetter	attrgetter(attr, ...) –> attrgetter object
defaultdict	defaultdict(default_factory[, ...]) –> dict with default factory
groupby(iterable[, key])	keys and groups from the iterable.
itemgetter	itemgetter(item, ...) –> itemgetter object
repeat(object [,times])	for the specified number of times.

SiteCfg

class ucca.convert.**SiteCfg**

Bases: object

Contains static configuration for conversion to/from the site XML.

Attributes Summary

EdgeConversion	
FALSE	version of site XML scheme which self adheres to
SchemeVersion	mapping of site XML tag attribute to layer1 edge tags.
TBD	values for True/False in the site XML (strings)
TRUE	
TagConversion	mapping of layer1.EdgeTags to site XML tag attributes.

Attributes Documentation

EdgeConversion = {'A': 'Participant', 'C': 'Center', 'D': 'aDverbial', 'E': 'Elaborato

FALSE = 'false'

version of site XML scheme which self adheres to

SchemeVersion = '1.0.4'

mapping of site XML tag attribute to layer1 edge tags.

TBD = 'To Be Defined'

values for True/False in the site XML (strings)

TRUE = 'true'

TagConversion = {'Center': 'C', 'Connector': 'N', 'Elaborator': 'E', 'Function': 'I
mapping of layer1.EdgeTags to site XML tag attributes.

SiteUtil

class ucca.convert.**SiteUtil**
Bases: object

Contains utility functions for converting to/from the site XML.

Functions: unescape: converts escaped characters to their original form. set_id: sets the Node ID (internal) attribute in the XML element. get_node: gets the node corresponding to the element given from

the mapping. If not found, returns None

set_node: writes the element site ID + node pair to the mapping

Methods Summary

get_node(e, mapp)
set_id(e, i)
set_node(e, n, mapp)
unescape(x)

Methods Documentation

static get_node(*e*, *mapp*)

static set_id(*e*, *i*)

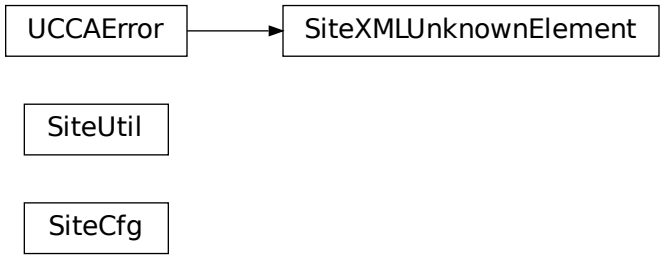
static set_node(*e*, *n*, *mapp*)

static unescape(*x*)

SiteXMLUnknownElement

exception ucca.convert.**SiteXMLUnknownElement**

Class Inheritance Diagram



2.1.4 ucca.core Module

This module encapsulate the basic elements of the UCCA annotation.

A UCCA annotation is practically a directed acyclic graph (DAG), which represents a *Passage* of text and its annotation. The annotation itself is divided into *Layer* objects, where in each layer *Node* objects are connected between themselves and to Nodes in other layers using *Edge* objects.

Functions

edge_id_orderkey(edge)	Key function which sorts Edges by its IDs (using *id_orderkey()*).
id_orderkey(node)	Key function which sorts by layer (string), then by unique ID (int).

edge_id_orderkey

ucca.core.**edge_id_orderkey**(*edge*)

Key function which sorts Edges by its IDs (using *id_orderkey()*).

Args: edge: *Edge* which we wish to sort according to the ID of its parent and children after using *id_orderkey()*.

Returns: a string with the layer and unique ID in such a way that sort will first order lexicography the layer ID then numerically the unique ID.

id_orderkey

ucca.core.**id_orderkey**(*node*)

Key function which sorts by layer (string), then by unique ID (int).

Args: node: *Node* which we will to sort according to its ID

Returns: a string with the layer and unique ID in such a way that sort will first order lexicography the layer ID then numerically the unique ID.

Classes

Category(tag[, slot, layer, parent])	when considering refinement layers, each edge can have multiple tags sorted in a certain hierarchy.
DuplicateIdError	Exception raised when trying to add an element with an existing ID.
Edge(root, parent, child[, tag, attrib])	Labeled edge between two *Node* objects in UCCA annotation graph.
FrozenPassageError	Exception raised when trying to modify a frozen *Passage*.
Layer(ID, root[, attrib, orderkey])	Group of similar *Node* objects in UCCA annotation graph.
MissingNodeError	Exception raised when trying to access a non-existent *Node*.
ModifyPassage(fn)	Decorator for changing a *Passage* or any member of it.
Node(ID, root, tag[, attrib, orderkey])	Labeled Node in UCCA annotation graph.
Passage(ID[, attrib])	An annotated text with UCCA annotation graph.
UCCAError	Base class for all UCCA package exceptions.
UnimplementedMethodError	Exception raised when trying to call a not-yet-implemented method.

Category

class ucca.core.**Category**(*tag*, *slot=None*, *layer=None*, *parent=None*)

Bases: object

when considering refinement layers, each edge can have multiple tags sorted in a certain hierarchy. for this reason, a category must include not only the tag information but also the layer and hierarchy information.

Attributes Summary

layer
parent
slot
tag

Methods Summary

to_xml()

Attributes Documentation

layer

parent

slot

tag

Methods Documentation

to_xml()

DuplicateIdError

exception ucca.core.**DuplicateIdError**
Exception raised when trying to add an element with an existing ID.

For each element, a unique ID must be assigned. If the ID of the new element is already present in the *Passage* in some way, this exception is raised.

Edge

class ucca.core.**Edge**(*root*, *parent*, *child*, *tag=None*, *attrib=None*)
Bases: object

Labeled edge between two *Node* objects in UCCA annotation graph.

An edge between Nodes in a *Passage* is a simple object; it is a directed edge whose ID is derived by the parent and child of the edge, it is mostly immutable except for its attributes, and it is labeled with the connection type between the Nodes. An edge can have multiple annotations, representing connection types of one or more layers.

Attributes: ID: ID of the Edge, constructed from the IDs of the two Nodes root: the Passage this object is linked with attrib: attribute dictionary of the Edge extra: temporary storage space for undocumented attributes and data tag: the string label of the Edge parent: the originating Node of the Edge child: the target Node of the Edge categories: a list of categories for this edge ID_FORMAT: format string which creates the ID of the Edge from

the IDs of the parent (first argument to the formatting string) and the child (second argument).

Attributes Summary

ID
ID_FORMAT
add
attrib
categories
child
parent
root
tag
tags

Methods Summary

equals(other, *[, recursive, ordered, ...])	Returns whether self and other are Edge-equals.

Attributes Documentation

ID

ID_FORMAT = '{}->{}'

add = functools.partial(<bound method ModifyPassage.__call__ of <ucca.core.ModifyPassa

attrib

categories

child

parent

root

tag

tags

Methods Documentation

equals(*other*, ***, *recursive=True*, *ordered=False*, *ignore_node=None*, *ignore_edge=None*)
Returns whether self and other are Edge-equals.

Edge-equality is determined by having the same tag and attributes. Recursive Edge-equality means that the Edges are equal, and their children are recursively Node-equal.

Parameters

- **other** – an Edge object to compare to
- **recursive** – whether to compare recursively, defaults to True
- **ordered** – if recursive, whether the children are Node-equivalent w.r.t order (see Node.equals())
- **ignore_node** – function that returns whether to ignore a given node
- **ignore_edge** – function that returns whether to ignore a given edge

Returns True iff the Edges are equal.

FrozenPassageError

exception ucca.core.**FrozenPassageError**
Exception raised when trying to modify a frozen *Passage*.

Layer

class ucca.core.**Layer**(*ID*, *root*, *attrib=None*, ***, *orderkey=<function id_orderkey>*)
Bases: object

Group of similar *Node* objects in UCCA annotation graph.

A Layer in UCCA annotation graph is a subgraph of the whole *Passage* annotation graph which consists of similar Nodes and *Edge* objects between them. The Nodes and the Layer itself has some formal definition for being grouped together.

Attributes: ID: ID of the Layer, must be alphanumeric. root: the Passage this object is linked with attrib: attribute dictionary of the Layer extra: temporary storage space for undocumented attributes and data orderkey: the key function for ordering the Nodes in the layer.

Note that it must rely only on the Nodes and/or Edges in the Layer. If it, for example, rely on Edges added between Nodes in the Layer and Nodes outside the Layer (hence, the Edges are not in the Layer) the order will not be updated (because the Layer object won't know that something has changed).

all: a list of all the Nodes which are part of this Layer heads: a list of all Nodes which have no incoming Edges in the subgraph

of the Layer (can have Edges from Nodes in other Layers).

Attributes Summary

ID
all
attrib
heads
orderkey
root

Methods Summary

equals(other, *[, ordered, ignore_node, ...])	Returns whether two Layer objects are equal.

Attributes Documentation

ID

all

attrib

heads

orderkey

root

Methods Documentation

equals(*other*, *, *ordered=False*, *ignore_node=None*, *ignore_edge=None*)

Returns whether two Layer objects are equal.

Layers are considered Layer-equal if their attribute dictionaries are equal and all their heads are recursively Node-equal. Ordered Layer-equality implies that the heads should be ordered the same for the Layers to be considered equal, and the Node-equality is ordered too.

Parameters

- **other** – the Layer object to compare to
- **ordered** – whether strict-order equality is used, defaults to False
- **ignore_node** – function that returns whether to ignore a given node

- **ignore_edge** – function that returns whether to ignore a given edge

Returns True iff self and other are Layer-equal.

MissingNodeError

exception ucca.core.**MissingNodeError**
Exception raised when trying to access a non-existent *Node*.

ModifyPassage

class ucca.core.**ModifyPassage**(*fn*)
Bases: object

Decorator for changing a *Passage* or any member of it.

This decorator is mandatory for anything which causes the elements in a *Passage* to change by adding or removing an element, or changing an attribute.

It validates that the Passage is not frozen before allowing the change.

The decorator can't be used for __init__ calls, as at the stage of the check there are no instance attributes to check. So in such cases, a function that binds the object created with the Passage should be decorated instead (and should be called after the instance attributes are set).

Attributes: fn: the function object to decorate

Methods Summary

__call__(*args, **kwargs)	Decorating functions which modify *Passage* elements.

Methods Documentation

__call__(**args*, ***kwargs*)
Decorating functions which modify *Passage* elements.

Parameters

- **args** – list of all arguments, assuming the first is the object which modifies *Passage*, and it has an attribute root which points to the Passage it is part of.
- **kwargs** – list of all keyword arguments

Returns The decorated function result.

Raises ***FrozenPassageError*** – if the *Passage* is frozen and can't be modified.

Node

class ucca.core.**Node**(*ID*, *root*, *tag*, *attrib=None*, ***, *orderkey=<function edge_id_orderkey>*)
Bases: object

Labeled Node in UCCA annotation graph.

A Node in *Passage* UCCA annotation is an vertex in the annotation graph, which may be an internal vertex or a leaf, and is labeled with a tag that specifies both the *Layer* it belongs to and it's ID in this Layer. It can have multiple children Nodes through *Edge* objects, and these children are ordered according to an internal order function.

Attributes:

ID: ID of the Node, constructed from the ID of the Layer it belongs to, a separator, and a unique alphanumeric ID in the layer.

root: the Passage this object is linked with attrib: attribute dictionary of the Node extra: temporary storage space for undocumented attributes and data tag: the string label of the Node layer: the Layer this Node belongs to incoming: a copy of the incoming Edges to this object outgoing: a copy of the outgoing Edges from this object parents: the Nodes which have incoming Edges to this object children: the Nodes which have outgoing Edges from this object orderkey: the key function for ordering the outgoing Edges ID_SEPARATOR: separator function between the Layer ID and the unique

Node ID in the complete ID of the Node. Mustn't be alphanumeric.

Attributes Summary

ID
ID_SEPARATOR
add
add_multiple
attrib
children
destroy
incoming
layer
orderkey
outgoing
parents
remove
root
tag

Methods Summary

equals(other, *[, recursive, ordered, ...])	Returns whether the self Node-equals other.
get_terminals(*args, **kwargs)	Returns a list of all terminals under the span of this Node.
iter([obj, method, duplicates, key])	Iterates the *Node* objects in the subtree of self.
missing_edges(other[, ignore_node])	Returns edges present in this node but missing in the other.

Attributes Documentation

ID

ID_SEPARATOR = '.'

add = functools.partial(<bound method ModifyPassage.__call__ of <ucca.core.ModifyPassa

add_multiple = functools.partial(<bound method ModifyPassage.__call__ of <ucca.core.Mo

attrib

children

destroy = functools.partial(<bound method ModifyPassage.__call__ of <ucca.core.ModifyP

incoming

layer

orderkey

outgoing

parents

remove = functools.partial(<bound method ModifyPassage.__call__ of <ucca.core.ModifyPa

root

tag

Methods Documentation

equals(*other*, *, *recursive=True*, *ordered=False*, *ignore_node=None*, *ignore_edge=None*)
Returns whether the self Node-equals other.

Node-equality is basically determined by self and other having the same tag and attributes. Recursive equality is achieved when all outgoing Edges are Edge-equal, and their children are recursively Node-equal as well. Ordered equality means that the outgoing Edges should be equivalent to each other w.r.t order (the first to the first etc.), while unordered equality means that each Edges are equivalent after being ordered with some determined order.

Parameters

- **other** – the Node object to compare to
- **recursive** – whether comparison is recursive, defaults to True.
- **ordered** – whether comparison should include strict ordering
- **ignore_node** – function that returns whether to ignore a given node
- **ignore_edge** – function that returns whether to ignore a given edge

Returns True iff the Nodes are equal in the terms given.

get_terminals(**args*, ***kwargs*)
Returns a list of all terminals under the span of this Node.

iter(*obj='nodes'*, *method='dfs'*, *duplicates=False*, *key=None*)
Iterates the Node objects in the subtree of self.

Parameters **obj** –

yield Node objects (use value "nodes", default) or Edge objects (use values "edges")

method: do breadth-first iteration (use value "bfs") or depth-first iteration (value "dfs", default).

duplicates: If True, may return the same object twice if it is encountered twice, because of the DAG structure which isn't necessarily a tree. If it is False, all objects will be yielded only the first time they are encountered. Defaults to False.

key: boolean function that filters the iterable items. key function takes one argument (the item) and returns True if it should be returned to the user. If an item isn't returned, its subtree is still iterated. Defaults to None (returns all items).

Yields: a *Node* or *Edge* object according to the iteration parameters.

missing_edges(*other*, *ignore_node=None*)

Returns edges present in this node but missing in the other.

Parameters

- **other** – the Node object to compare to
- **ignore_node** – function that returns whether to ignore a given node

Returns List of edges present in this node but missing in the other.

Passage

class ucca.core.**Passage**(*ID*, *attrib=None*)

Bases: object

An annotated text with UCCA annotation graph.

A Passage is an object representing a text annotated with UCCA annotation. UCCA annotation is a directed acyclic graph of *Node* and *Edge* objects grouped into *Layer* objects.

Attributes: ID: ID of the Passage root: simply self, for API similarity with other UCCA objects attrib: attribute dictionary of the Passage extra: temporary storage space for undocumented attributes and data layers: all Layers of the Passage, no order guaranteed nodes: dictionary of ID-node pairs for all the nodes in the Passage frozen: indicates whether the Passage can be modified or not, boolean.

Attributes Summary

ID
attrib
categories
layers
nodes
refined_categories
root

Methods Summary

by_id(ID)	Returns a Node whose ID is given.
copy([layers])	Copies the Passage and specified layers to a new object.
equals(other, *[, ordered, ignore_node, ...])	Returns whether two passages are equivalent.
layer(ID)	Returns the *Layer* object whose ID is given.
missing_nodes(other[, ignore_node, ignore_edge])	Returns nodes present in this passage but missing in the other.

Attributes Documentation

ID

attrib

categories

layers

nodes

refined_categories

root

Methods Documentation

by_id(*ID*)
Returns a Node whose ID is given.

Parameters **ID** – ID string

Returns The node.Node object whose ID matches

Raises **KeyError** – if no Node with this ID is found

copy(*layers=None*)
Copies the Passage and specified layers to a new object.

The main "building block" of copying is the Layer, so copying is truly copying the Passage attributes (attrib, extra, ID, frozen) and creating the equivalent layers (each layer for itself).

Parameters **layers** – sequence of layer IDs to copy to the new object. If None, all layers will be copied.

Returns A new Passage object.

Raises **KeyError** – if a given layer ID doesn't exist. UnimplementedMethodError if copying for a layer is unimplemented.

equals(*other*, *, *ordered=False*, *ignore_node=None*, *ignore_edge=None*)
Returns whether two passages are equivalent.

Passage-equivalence is determined by having the same attributes and all layers (according to ID) are Layer-equivalent.

Parameters

- **other** – the Passage object to compare to
- **ordered** – is Layer-equivalency should be ordered (see there)
- **ignore_node** – function that returns whether to ignore a given node
- **ignore_edge** – function that returns whether to ignore a given edge

Returns True iff self is Passage-equivalent to other.

layer(*ID*)
Returns the *Layer* object whose ID is given.

Parameters **ID** – ID of the Layer requested.

Raises **KeyError** – if no Layer with this ID is present

missing_nodes(*other*, *ignore_node=None*, *ignore_edge=None*)
Returns nodes present in this passage but missing in the other.

Parameters

- **other** – the Passage object to compare to
- **ignore_node** – function that returns whether to ignore a given node
- **ignore_edge** – function that returns whether to ignore a given edge

Returns List of nodes present in this passage but missing in the other.

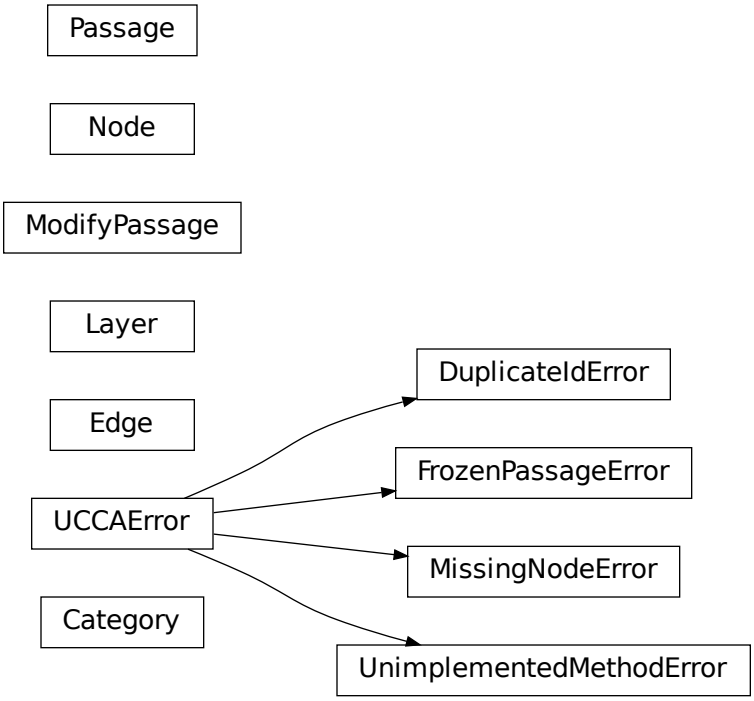
UCCAError

exception ucca.core.**UCCAError**
Base class for all UCCA package exceptions.

UnimplementedMethodError

exception ucca.core.**UnimplementedMethodError**
Exception raised when trying to call a not-yet-implemented method.

Class Inheritance Diagram



2.1.5 ucca.diffutil Module

Functions

diff_passages(true_passage, pred_passage[, ...])	Debug method to print missing or mistaken attributes, nodes and edges
passage2file(passage, filename[, indent, binary])	Writes a UCCA passage as a standard XML file or a binary pickle :param passage: passage object to write :param filename: file name to write to :param indent: whether to indent each line :param binary: whether to write pickle format (or XML)

diff_passages

ucca.diffutil.**diff_passages**(*true_passage*, *pred_passage*, *write=False*)

Debug method to print missing or mistaken attributes, nodes and edges

2.1.6 ucca.evaluation Module

The evaluation library for UCCA layer 1. v1.4 2016-12-25: move common Fs to root before evaluation 2017-01-04: flatten centers, do not add 1 (for root) to mutual 2017-01-16: fix bug in moving common Fs 2018-04-12: exclude punctuation nodes regardless of edge tag 2018-12-11: fix another bug in moving common Fs 2019-01-22: support multiple categories per edge 2019-11-29: evaluate implicit nodes too (by their parent's yield)

Functions

create_passage_yields(p, *args[, tags])	**param p** passage to find terminal yields of
evaluate(guessed, ref[, converter, verbose, ...])	Compare two passages and return requested diagnostics and scores, possibly printing them too.
expand_equivalents(tag_set)	Returns a set of all the tags in the tag set or those equivalent to them :param tag_set: set of tags (strings) to expand
get_by_names([names])	
get_text(p, positions)	
get_yield(unit)	
move_functions(p1, p2)	Move any common Fs to the root
print_tags_and_text(p, yield_tags)	

evaluate

ucca.evaluation.**evaluate**(*guessed*, *ref*, *converter=None*, *verbose=False*, *constructions={'implicit': <ucca.constructions.Construction object>, 'primary': <ucca.constructions.Construction object>, 'remote': <ucca.constructions.Construction object>}*, *units=False*, *fscore=True*, *errors=False*, *normalize=True*, *eval_type=None*, *ref_yield_tags=None*, ***kwargs*)

Compare two passages and return requested diagnostics and scores, possibly printing them too. NOTE: since

normalize=True by default, this method is destructive: it modifies the given passages before evaluation. :param guessed: Passage object to evaluate :param ref: reference Passage object to compare to :param converter: optional function to apply to passages before evaluation :param verbose: whether to print the results :param constructions: names of construction types to include in the evaluation :param units: whether to evaluate common units :param fscore: whether to compute precision, recall and f1 score :param errors: whether to print the mistakes :param normalize: flatten centers and move common functions to root before evaluation - modifies passages :param eval_type: specific evaluation type(s) to limit to :param ref_yield_tags: reference passage for fine-grained evaluation :return: Scores object

expand_equivalents

ucca.evaluation.**expand_equivalents**(*tag_set*)

Returns a set of all the tags in the tag set or those equivalent to them :param tag_set: set of tags (strings) to expand

get_text

ucca.evaluation.**get_text**(*p*, *positions*)

get_yield

ucca.evaluation.**get_yield**(*unit*)

move_functions

ucca.evaluation.**move_functions**(*p1*, *p2*)

Move any common Fs to the root

print_tags_and_text

ucca.evaluation.**print_tags_and_text**(*p*, *yield_tags*)

Classes

Counter(**kwds)	Dict subclass for counting hashable items.
EdgeTags	Layer 1 Edge tags.
Evaluator(verbose, constructions, units, ...)	
EvaluatorResults(results[, default])	
NodeTags	Layer 1 Node tags.
OrderedDict	Dictionary that remembers insertion order
Scores(evaluator_results[, name, ...])	
SummaryStatistics(num_matches, ... [, errors])	
attrgetter	attrgetter(attr, ...) –> attrgetter object
groupby(iterable[, key])	keys and groups from the iterable.

Evaluator

class ucca.evaluation.**Evaluator**(*verbose*, *constructions*, *units*, *fscore*, *errors*)

Bases: object

Methods Summary

find_mutuals(m1, m2, eval_type, mutual_tags)	
get_scores(p1, p2, eval_type[, r])	prints the relevant statistics and f-scores.

Methods Documentation

static find_mutuals(*m1*, *m2*, *eval_type*, *mutual_tags*, *counter=None*)

get_scores(*p1*, *p2*, *eval_type*, *r=None*)

prints the relevant statistics and f-scores. eval_type can be 'unlabeled', 'labeled' or 'weak_labeled'. calculates a set of all the yields such that both passages have a unit with that yield. :param p1: passage to compare :param p2: reference passage object :param eval_type: evaluation type to use, out of EVAL_TYPES 1. UNLABELED: it doesn't matter what labels are there. 2. LABELED: also requires tag match (if there are multiple units with the same yield, requires one match) 3. WEAK_LABELED: also requires weak tag match (if there are multiple units with the same yield,

requires one match)

Parameters **r** – reference passage for fine-grained evaluation

Returns EvaluatorResults object if self.fscore is True, otherwise None

EvaluatorResults

class ucca.evaluation.**EvaluatorResults**(*results*, *default=None*)

Bases: object

Methods Summary

aggregate(results)	**param results** iterable of EvaluatorResults
aggregate_default()	Aggregate primary and remote SummaryStatistics in this EvaluatorResults instance :return: SummaryStatistics object representing aggregation over primary and remote
print(**kwargs)	
print_confusion_matrix([prefix, sep, as_table])	

Methods Documentation

classmethod aggregate(*results*)

Parameters **results** – iterable of EvaluatorResults

Returns new EvaluatorResults with aggregates scores

aggregate_default()
Aggregate primary and remote SummaryStatistics in this EvaluatorResults instance :return: SummaryStatistics object representing aggregation over primary and remote

print(***kwargs*)

print_confusion_matrix(*prefix=None*, *sep=None*, *as_table=False*, ***kwargs*)

Scores

class ucca.evaluation.**Scores**(*evaluator_results*, *name=None*, *evaluation_format=None*)
Bases: object

Methods Summary

aggregate(scores)	Aggregate multiple Scores instances :param scores: iterable of Scores :return: new Scores with aggregated scores
average_f1([mode])	Calculate the average F1 score across primary and remote edges :param mode: LABELED, UNLABELED or WEAK_LABELED :return: a single number, the average F1
field_titles([constructions, eval_type, counts])	
fields([eval_type, counts])	
print([eval_type])	
print_confusion_matrix(*args[, eval_type])	
titles([eval_type, counts])	

Methods Documentation

static aggregate(*scores*)
Aggregate multiple Scores instances :param scores: iterable of Scores :return: new Scores with aggregated scores

average_f1(*mode='labeled'*)
Calculate the average F1 score across primary and remote edges :param mode: LABELED, UNLABELED or WEAK_LABELED :return: a single number, the average F1

static field_titles(*constructions={'implicit': <ucca.constructions.Construction object>, 'primary': <ucca.constructions.Construction object>, 'remote': <ucca.constructions.Construction object>}*, *eval_type='labeled'*, *counts=False*)

fields(*eval_type='labeled'*, *counts=False*)

print(*eval_type=None*, ***kwargs*)

print_confusion_matrix(**args*, *eval_type=None*, ***kwargs*)

titles(*eval_type='labeled'*, *counts=False*)

SummaryStatistics

class ucca.evaluation.**SummaryStatistics**(*num_matches*, *num_only_guessed*, *num_only_ref*, *errors=None*)

Bases: object

Methods Summary

aggregate(stats)	**param stats** iterable of SummaryStatistics
print(**kwargs)	

Methods Documentation

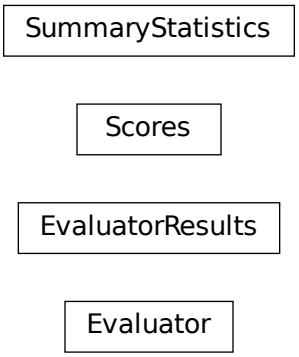
classmethod aggregate(*stats*)

Parameters **stats** – iterable of SummaryStatistics

Returns new SummaryStatistics with aggregated scores

print(***kwargs*)

Class Inheritance Diagram



2.1.7 ucca.ioutil Module

Input/output utility functions for UCCA scripts.

Functions

contextmanager(func)	@contextmanager decorator.
external_write_mode(*args, **kwargs)	
file2passage(filename)	Opens a file and returns its parsed Passage object Tries to read both as a standard XML file and as a binary pickle :param filename: file name to write to
from_text(text[, passage_id, tokenized, ...])	Converts from tokenized strings to a Passage object.
gen_files(files_and_dirs)	**param files_and_dirs** iterable of files and/or directories to look in
get_passages(filename_patterns, **kwargs)	
get_passages_with_progress_bar(filename_patterns)	
glob(pathname, *[, recursive])	Return a list of paths matching a pathname pattern.
passage2file(passage, filename[, indent, binary])	Writes a UCCA passage as a standard XML file or a binary pickle :param passage: passage object to write :param filename: file name to write to :param indent: whether to indent each line :param binary: whether to write pickle format (or XML)
read_files_and_dirs(files_and_dirs[, ...])	**param files_and_dirs** iterable of files and/or directories to look in
resolve_patterns(filename_patterns)	
split2segments(passage, is_sentences[, ...])	Split passage to sub-passages :param passage: Passage object :param is_sentences: if True, split to sentences; otherwise, paragraphs :param remarks: Whether to add remarks with original node IDs :param lang: language to use for sentence splitting model :param ids: optional iterable of ids to set passage IDs for each split :return: sequence of passages
to_text(passage[, sentences, lang])	Converts from a Passage object to tokenized strings.
write_passage(passage[, output_format, ...])	Write a given UCCA passage in any format.

external_write_mode

ucca.ioutil.**external_write_mode**(**args*, ***kwargs*)

gen_files

ucca.ioutil.**gen_files**(*files_and_dirs*)

Parameters **files_and_dirs** – iterable of files and/or directories to look in

Returns all files given, plus any files directly under any directory given

get_passages

ucca.ioutil.**get_passages**(*filename_patterns*, ***kwargs*)

get_passages_with_progress_bar

ucca.ioutil.**get_passages_with_progress_bar**(*filename_patterns*, *desc=None*, ***kwargs*)

read_files_and_dirs

ucca.ioutil.**read_files_and_dirs**(*files_and_dirs*, *sentences=False*, *paragraphs=False*, *converters=None*, *lang='en'*, *attempts=3*, *delay=5*)

Parameters

- **files_and_dirs** – iterable of files and/or directories to look in
- **sentences** – whether to split to sentences
- **paragraphs** – whether to split to paragraphs
- **converters** – dict of input format converters to use based on the file extension
- **lang** – language to use for tokenization model
- **attempts** – number of times to try reading a file before giving up
- **delay** – number of seconds to wait before subsequent attempts to read a file

Returns lazy-loaded passages from all files given, plus any files directly under any directory given

resolve_patterns

ucca.ioutil.**resolve_patterns**(*filename_patterns*)

write_passage

ucca.ioutil.**write_passage**(*passage*, *output_format=None*, *binary=False*, *outdir='.'*, *prefix=''*, *converter=None*, *verbose=True*, *append=False*, *basename=None*)

Write a given UCCA passage in any format. :param passage: Passage object to write :param output_format: filename suffix (if given "ucca", suffix will be ".pickle" or ".xml" depending on 'binary') :param binary: save in pickle format with ".pickle" suffix :param outdir: output directory, should exist already :param prefix: string to prepend to output filename :param converter: function to apply to passage before saving (if output_format is not "ucca"/"pickle"/"xml"),

returning iterable of strings, each corresponding to an output line

Parameters

- **verbose** – print "Writing passage" message
- **append** – if using converter, append to output file rather than creating a new file
- **basename** – use this instead of 'passage.ID' for the output filename

Returns path of created output file

Classes

LazyLoadedPassages(files[, sentences, ...])	Iterable interface to Passage objects that loads files on-the-go and can be iterated more than once
ParseError	
Passage(ID[, attrib])	An annotated text with UCCA annotation graph.
chain	chain(*iterables) –> chain object
defaultdict	defaultdict(default_factory[, ...]) –> dict with default factory
filterfalse	filterfalse(function or None, sequence) –> filterfalse object
tqdm([iterable, desc, total, leave, file, ...])	Decorate an iterable object, returning an iterator which acts exactly like the original iterable, but prints a dynamically updating progressbar every time a value is requested.

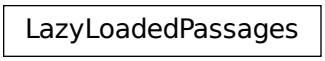
LazyLoadedPassages

class ucca.ioutil.**LazyLoadedPassages**(*files*, *sentences=False*, *paragraphs=False*, *converters=None*, *lang='en'*, *attempts=3*, *delay=5*)

Bases: object

Iterable interface to Passage objects that loads files on-the-go and can be iterated more than once

Class Inheritance Diagram



2.1.8 ucca.layer0 Module

Encapsulates all word and punctuation symbols layer.

Layer 0 is the basic layer for all the UCCA annotation, as it includes the actual words and punctuation marks found in the core.Passage.

Layer 0 has only one type of node, *Terminal*. This is a subtype of core.Node, and can have one of two tags: Word or Punctuation.

Functions

is_punct(node)	Returns whether the unit is a layer0 punctuation (for all Units).

is_punct

ucca.layer0.**is_punct**(*node*)
Returns whether the unit is a layer0 punctuation (for all Units).

Classes

Layer0(root[, attrib])	Represents the *Terminal* objects layer.
NodeTags	
Terminal(ID, root, tag[, attrib, orderkey])	Layer 0 Node type, represents a word or a punctuation mark.

Layer0

class ucca.layer0.**Layer0**(*root*, *attrib=None*)
Bases: *ucca.core.Layer*

Represents the *Terminal* objects layer.

Attributes: words: a tuple of only the words (not punctuation) Terminals, ordered pairs: a tuple of (position, terminal) tuples of all Terminals, ordered

Attributes Summary

pairs	
words	

Methods Summary

add_terminal(text, punct[, paragraph])	Adds the next Terminal at the next available position.
by_position(pos)	Returns the Terminals at the position given.
copy(other_passage)	Creates a copied Layer0 object and Terminals in other_passage.
doc(paragraph)	
docs([num_paragraphs])	

Attributes Documentation

pairs

words

Methods Documentation

add_terminal(*text*, *punct*, *paragraph=1*)
Adds the next Terminal at the next available position.

Creates a *Terminal* object with the next position, assuming that all positions are filled (no holes).

Parameters

- **text** – the text of the Terminal
- **punct** – boolean, whether it's a punctuation mark
- **paragraph** – paragraph number, defaults to 1

Returns the created Terminal

Raises ***DuplicateIdError*** – if trying to add an already existing Terminal, caused by unordered Terminal positions in the layer

by_position(*pos*)

Returns the Terminals at the position given.

Parameters **pos** – the position of the Terminal object

Returns the Terminal in this position

Raises **IndexError** – if the position is out of bounds

copy(*other_passage*)

Creates a copied Layer0 object and Terminals in other_passage.

Parameters **other_passage** – the Passage to copy self to

doc(*paragraph*)

docs(*num_paragraphs=1*)

NodeTags

class ucca.layer0.**NodeTags**

Bases: object

Attributes Summary

Punct
Word

Attributes Documentation

Punct = 'Punctuation'

Word = 'Word'

Terminal

class ucca.layer0.**Terminal**(*ID*, *root*, *tag*, *attrib=None*, ***, *orderkey=<function edge_id_orderkey>*)

Bases: *ucca.core.Node*

Layer 0 Node type, represents a word or a punctuation mark.

Terminals are core.Node objects which represent a word or a punctuation mark in the core.Passage object. They are immutable, as they shouldn't be changed throughout their use and have no children. Hence, they can be compared and hashed, unlike other core.Node subclasses.

Attributes: ID: the unique ID of each Terminal is its global position in the Passage, e.g. ID=0.4 is the 4th Terminal in the `Passage`. tag: from NodeTags layer: '0' (LAYER_ID) attrib: returns a copy of the attribute dictionary, so changing it

will not affect the Terminal object

text: text of the Terminal, whether punctuation or a word position: global position of the Terminal in the passage, starting at 1 paragraph: which paragraph the Terminal belongs to, starting at 1 para_pos: the position of the Terminal in the paragraph,

starting at 1 (per paragraph).

punct: whether the Terminal is a punctuation mark (boolean)

Attributes Summary

attrib
para_pos
paragraph
position
punct
text
tok

Methods Summary

add(*args, **kwargs)	partial(func, *args, **keywords) - new function with partial application of the given arguments and keywords.
equals(other, *[, ordered])	Equals if the Terminals are of the same Layer, tag, position & text.
get_annotation(attr[, as_array])	
get_terminals([punct])	Returns a list containing just this Terminal.
remove(*args, **kwargs)	partial(func, *args, **keywords) - new function with partial application of the given arguments and keywords.

Attributes Documentation

attrib

para_pos

paragraph

position

punct

text

tok

Methods Documentation

add (**args*, ***kwargs*)
partial(func, *args, **keywords) - new function with partial application of the given arguments and keywords.

equals (*other*, *, *ordered=False*, ***kwargs*)
Equals if the Terminals are of the same Layer, tag, position & text.

Parameters

- **other** – another Terminal to equal to
- **ordered** – unused, here for API conformity.

Returns True iff the two Terminals are equal.

get_annotation (*attr*, *as_array=False*)

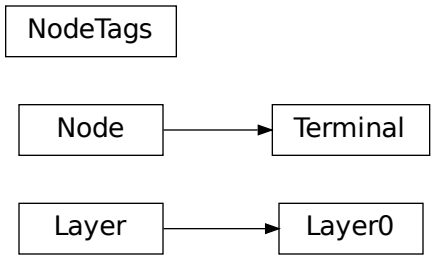
get_terminals (*punct=True*, **args*, ***kwargs*)
Returns a list containing just this Terminal.

Parameters **punct** – whether to include punctuation Terminals, defaults to True

Returns a list of `layer0`.Terminal objects

remove (**args*, ***kwargs*)
partial(func, *args, **keywords) - new function with partial application of the given arguments and keywords.

Class Inheritance Diagram



2.1.9 ucca.layer1 Module

Describes the foundational level elements (layer 1) of the UCCA annotation.

Layer 1 is the foundational layer of UCCA, whose Nodes and Edges represent scene objects and relations. The basic building blocks of this layer are the FNode, which is a participant in a scene relation (including the relation itself), and the various Edges between these Nodes, which represent the type of relation between the Nodes.

Classes

EdgeTags	Layer 1 Edge tags.
FoundationalNode(ID, root, tag[, attrib, ...])	The basic building block of UCCA annotation, represents semantic units.
Layer1(root[, attrib, orderkey])	
Linkage(ID, root, tag[, attrib, orderkey])	A Linkage between parallel scenes.
MissingRelationError	Exception raised when a required edge is not present.
NodeTags	Layer 1 Node tags.
PunctNode(ID, root, tag[, attrib, orderkey])	Encapsulates punctuation `layer0`.Terminal objects.

EdgeTags

class ucca.layer1.**EdgeTags**

Bases: object

Layer 1 Edge tags.

Attributes Summary

Adverbial
Center
Connector
Elaborator
Function
Ground
LinkArgument
LinkRelation
Linker
ParallelScene
Participant
Process
Punctuation
Quantifier
Relator
State
Terminal
Time
Unanalyzable
Uncertain

Attributes Documentation

Adverbial = 'D'

Center = 'C'

Connector = 'N'

Elaborator = 'E'

Function = 'F'

```
Ground = 'G'
LinkArgument = 'LA'
LinkRelation = 'LR'
Linker = 'L'
ParallelScene = 'H'
Participant = 'A'
Process = 'P'
Punctuation = 'U'
Quantifier = 'Q'
Relator = 'R'
State = 'S'
Terminal = 'Terminal'
Time = 'T'
Unanalyzable = 'UNA'
Uncertain = 'UNC'
```

FoundationalNode

class ucca.layer1.**FoundationalNode**(*ID*, *root*, *tag*, *attrib=None*, ***, *orderkey=<function edge_id_orderkey>*)

Bases: *ucca.core.Node*

The basic building block of UCCA annotation, represents semantic units.

Each FoundationalNode (FNode for short) represents a semantic unit in the text, with relations to other semantic units. In essence, the FNodes form a tree of annotation, when remote units are ignored. This means that each FNode has exactly one FNode parent, and for completeness, there is also a "Passage Head" FNode which is the FNode parent of all parallel scenes and linkers in the top-level of the annotation.

Remote units are FNodes which are shared between two or more different FNodes, and hence have two FNode parents (participate in two relations). In such cases there is only one FNode parent, as the other Edges to parents are marked with the 'remote' attribute (set to True).

Implicit Nodes are ones which aren't mentioned in the text, and hence doesn't have any Terminal units in their span. In such cases, they will have an 'implicit' attribute set to True, and will take the position -1 (both start and end positions).

Attributes: participants: adverbials: connector: grounds: elaborators: centers: linkers: parallel_scenes: functions: punctuation: terminals:

a list of all FNodes under self whose edge tag is one of these types.

process: state: time: relator:

Returns the FNode under self whose edge tag is one of these types, or None in case it isn't found.

start_position: end_position:

start/end position of the first/last terminal in the span of the FNode, without counting in remote FNodes. If the FNode is implicit or have no Terminals for some reason, returns -1 (both).

fparent: the FNode parent (FNode with incoming Edge, not remote) of this FNode. There is exactly one for each FNode except the Passage head, which returns None.

ftag: the tag of the Edge connecting the fparent (as described above) with this FNode

discontiguous: whether this FNode has continuous Terminals or not

Attributes Summary

adverbials
centers
connector
discontiguous
elaborators
end_position
fparent
ftag
ftags
functions
grounds
linkers
parallel_scenes
participants
process
punctuation
quantifiers
relator
start_position
state
terminals
times

Methods Summary

get_sequences()	
get_terminals([punct, remotes, visited])	Returns a list of all terminals under the span of this FoundationalNode.
get_top_scene()	Returns the top-level scene this FNode is within, or None
is_scene()	
to_text()	Returns the text in the span of self, separated by spaces.

Attributes Documentation

adverbials

centers

connector

discontiguous

elaborators

end_position

fparent

ftag

ftags

functions

grounds

linkers

parallel_scenes

participants

process

punctuation

quantifiers

relator

start_position

state

terminals

times

Methods Documentation

get_sequences()

get_terminals(*punct=True*, *remotes=False*, *visited=None*)
Returns a list of all terminals under the span of this FoundationalNode. :param punct: whether to include punctuation Terminals, defaults to True :param remotes: whether to include Terminals from remote FoundationalNodes, defaults to false :param visited: used to detect cycles :return: a list of `layer0`.Terminal objects

get_top_scene()
Returns the top-level scene this FNode is within, or None

is_scene()

to_text()
Returns the text in the span of self, separated by spaces.

Layer1

class ucca.layer1.**Layer1**(*root*, *attrib=None*, *, *orderkey=<function id_orderkey>*)
Bases: *ucca.core.Layer*

Attributes Summary

top_linkages	
top_scenes	

Methods Summary

add_fnode(parent, tag, *[, implicit])	
add_fnode_multiple(parent, edge_categories, *)	Adds a new FNode whose parent and Edge tag are given.
add_linkage(relation, *args)	Adds a Linkage between the link relation and the linked arguments.
add_punct(parent, terminal[, layer, slot, ...])	Adds a PunctNode as the child of parent and the Terminal under it.
add_remote(parent, tag, child)	
add_remote_multiple(parent, edge_categories, ...)	Adds a new core.Edge with remote attribute between the nodes.
next_id()	Returns the next available ID string for this layer.

Attributes Documentation

top_linkages

top_scenes

Methods Documentation

add_fnode(*parent*, *tag*, *, *implicit=False*)

add_fnode_multiple(*parent*, *edge_categories*, *, *implicit=False*, *edge_attrib=None*)

Adds a new FNode whose parent and Edge tag are given.

Parameters

- **parent** – the FNode which will be the parent of the new FNode. If the parent is None, adds under the layer head FNode.
- **edge_categories** – list of categories on the Edge between the parent and the new FNode.
- **implicit** – whether to set the new FNode as implicit (default False)
- **edge_attrib** – Keyword only, dictionary of attributes to be passed to the Edge initializer.

Returns the newly created FNode

:raise core.FrozenPassageError if the Passage is frozen

add_linkage(*relation*, **args*)

Adds a Linkage between the link relation and the linked arguments.

Linkage objects are all heads and have no parents.

Parameters

- **relation** – the link relation FNode.
- **args** – any number (at least 1) of linkage arguments FNodes.

Returns the newly created Linkage

:raise core.FrozenPassageError if the Passage is frozen.

add_punct(*parent*, *terminal*, *layer=None*, *slot=None*, *edge_attrib=None*)
Adds a PunctNode as the child of parent and the Terminal under it.

Parameters

- **parent** – the parent of the newly created PunctNode. If None, adds under rhe layer head FNode.
- **terminal** – the punctuation Terminal we want to put under parent.
- **edge_attrib** – Keyword only, dictionary of attributes to be passed to the Edge initializer.

Returns the newly create PunctNode.

:raise core.FrozenPassageError if the Passage is frozen.

add_remote(*parent*, *tag*, *child*)

add_remote_multiple(*parent*, *edge_categories*, *child*, *edge_attrib=None*)
Adds a new core.Edge with remote attribute between the nodes.

Parameters

- **parent** – the parent of the remote Edge
- **edge_categories** – list of categories of the Edge
- **child** – the child of the remote Edge
- **edge_attrib** – Keyword only, dictionary of attributes to be passed to the Edge initializer.

:raise core.FrozenPassageError if the Passage is frozen

next_id()
Returns the next available ID string for this layer.

Linkage

class ucca.layer1.**Linkage**(*ID*, *root*, *tag*, *attrib=None*, *, *orderkey=<function edge_id_orderkey>*)
Bases: *ucca.core.Node*

A Linkage between parallel scenes.

A Linkage object represents a connection between two parallel scenes. The semantic type of the link is not determined in this object, but the *FoundationalNode* of linkage is referred as the link relation, and the linked scenes are referred to as the arguments.

Most cases will have two arguments, but some constructions have 1 or 3+ arguments, depending on the semantic connection.

Attributes: relation: FoundationalNode of the relation words. arguments: list of FoundationalNodes of the relation participants.

Attributes Summary

arguments
relation

Attributes Documentation

arguments

relation

MissingRelationError

exception ucca.layer1.**MissingRelationError**
Exception raised when a required edge is not present.

NodeTags

class ucca.layer1.**NodeTags**
Bases: object

Layer 1 Node tags.

Attributes Summary

Foundational
Linkage
Punctuation

Attributes Documentation

Foundational = 'FN'

Linkage = 'LKG'

Punctuation = 'PNCT'

PunctNode

class ucca.layer1.**PunctNode**(*ID*, *root*, *tag*, *attrib=None*, ***, *orderkey=<function edge_id_orderkey>*)
Bases: *ucca.layer1.FoundationalNode*

Encapsulates punctuation layer0.Terminal objects.

Attributes:

terminals: return the layer0.Terminal objects encapsulated by this Node in a list (at least one, usually not more than 1).

start_position: end_position:

start/end position of the first/last terminal in the span of the PunctNode.

Attributes Summary

terminals	

Methods Summary

add(edge_tag, node, *[, edge_attrib])	partial(func, *args, **keywords) - new function with partial application of the given arguments and keywords.
get_terminals([punct])	Returns a list of all terminals under the span of this PunctNode.

Attributes Documentation

terminals

Methods Documentation

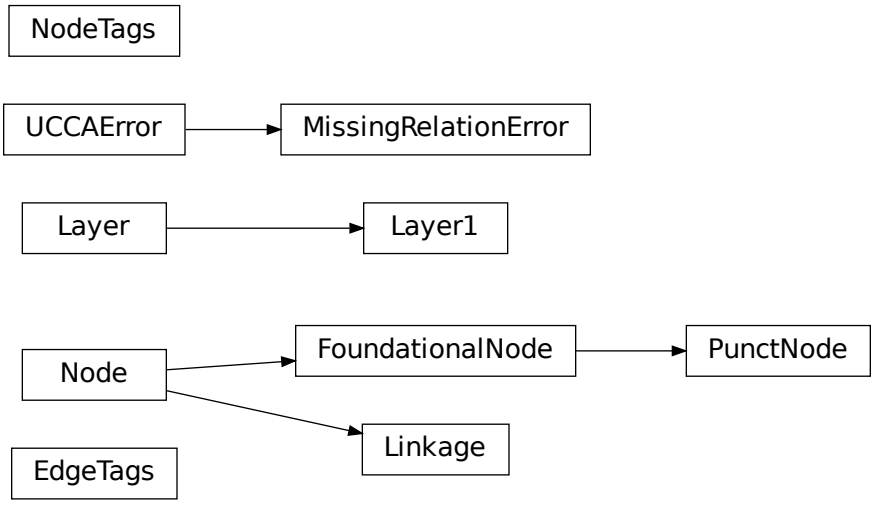
add(*edge_tag*, *node*, *, *edge_attrib=None*)
partial(func, *args, **keywords) - new function with partial application of the given arguments and keywords.

get_terminals(*punct=True*, **args*, ***kwargs*)
Returns a list of all terminals under the span of this PunctNode.

Parameters **punct** – whether to include punctuation Terminals, defaults to True

Returns a list of `layer0`.Terminal objects

Class Inheritance Diagram



2.1.10 ucca.normalization Module

Functions

attach_punct(l0, l1)	
attach_terminals(l0, l1)	
copy_edge(edge[, parent, child, tag, attrib])	
destroy(node_or_edge)	
detach_punct(l1)	
flatten_centers(node)	Whenever there are Cs inside Cs, remove the external C.
flatten_functions(node)	Whenever there is an F as an only child, remove it.
flatten_participants(node)	Whenever there is an A as an only child, remove it.
fparent(node_or_edge)	
lowest_common_ancestor(*nodes)	
move_elements(node, tags, parent_tags[, forward])	
move_scene_elements(node)	
move_sub_scene_elements(node)	
nearest_parent(l0, *terminals)	
nearest_word(l0, position, step)	
normalize(passage[, extra])	
normalize_node(node, l1, extra)	
reattach_punct(l0, l1)	
reattach_terminals(l0, l1)	
remove(parent, child)	

Continued on next page

Table 51 – continued from previous page

remove_unmarked_implicits(node)
replace_center(edge)
replace_edge_tags(node)
separate_scenes(node, l1[, top_level])
split_coordinated_main_rel(node, l1)
traverse_up_centers(node)

attach_punct

ucca.normalization.**attach_punct**(*l0*, *l1*)

attach_terminals

ucca.normalization.**attach_terminals**(*l0*, *l1*)

copy_edge

ucca.normalization.**copy_edge**(*edge*, *parent=None*, *child=None*, *tag=None*, *attrib=None*)

destroy

ucca.normalization.**destroy**(*node_or_edge*)

detach_punct

ucca.normalization.**detach_punct**(*l1*)

flatten_centers

ucca.normalization.**flatten_centers**(*node*)
Whenever there are Cs inside Cs, remove the external C. Whenever there is a C as an only child, remove it.

flatten_functions

ucca.normalization.**flatten_functions**(*node*)
Whenever there is an F as an only child, remove it. If an F has non-terminal children, move them up.

flatten_participants

ucca.normalization.**flatten_participants**(*node*)
Whenever there is an A as an only child, remove it. If there is an implicit A in a scene without a main relation, remove it.

fparent

ucca.normalization.**fparent**(*node_or_edge*)

lowest_common_ancestor

ucca.normalization.**lowest_common_ancestor**(**nodes*)

move_elements

ucca.normalization.**move_elements**(*node*, *tags*, *parent_tags*, *forward=True*)

move_scene_elements

ucca.normalization.**move_scene_elements**(*node*)

move_sub_scene_elements

ucca.normalization.**move_sub_scene_elements**(*node*)

nearest_parent

ucca.normalization.**nearest_parent**(*l0*, **terminals*)

nearest_word

ucca.normalization.**nearest_word**(*l0*, *position*, *step*)

normalize

ucca.normalization.**normalize**(*passage*, *extra=False*)

normalize_node

ucca.normalization.**normalize_node**(*node*, *l1*, *extra*)

reattach_punct

ucca.normalization.**reattach_punct**(*l0*, *l1*)

reattach_terminals

ucca.normalization.**reattach_terminals**(*l0*, *l1*)

remove

ucca.normalization.**remove**(*parent*, *child*)

remove_unmarked_implicits

ucca.normalization.**remove_unmarked_implicits**(*node*)

replace_center

ucca.normalization.**replace_center**(*edge*)

replace_edge_tags

ucca.normalization.**replace_edge_tags**(*node*)

separate_scenes

ucca.normalization.**separate_scenes**(*node*, *l1*, *top_level=False*)

split_coordinated_main_rel

ucca.normalization.**split_coordinated_main_rel**(*node*, *l1*)

traverse_up_centers

ucca.normalization.**traverse_up_centers**(*node*)

2.1.11 ucca.textutil Module

Utility functions for UCCA package.

Functions

annotate(passage, *args, **kwargs)	Run spaCy pipeline on the given passage, unless already annotated :param passage: Passage object, whose layer 0 nodes will be added entries in the 'extra' dict

Continued on next page

Table 52 – continued from previous page

annotate_all(passages[, replace, as_array, ...])	Run spaCy pipeline on the given passages, unless already annotated :param passages: iterable of Passage objects, whose layer 0 nodes will be added entries in the 'extra' dict :param replace: even if a given passage is already annotated, replace with new annotation :param as_array: instead of adding 'extra' entries to each terminal, set layer 0 extra["doc"] to array of ids :param as_extra: set 'extra' entries to each terminal :param as_tuples: treat input as tuples of (passage text, context), and return context for each passage as-is :param lang: optional two-letter language code, will be overridden if passage has "lang" attrib :param vocab: optional dictionary of vocabulary IDs to string values, to avoid loading spaCy model :param verbose: whether to print annotated text :return: generator of annotated passages, which are actually modified in-place (same objects as input)
annotate_as_tuples(passages[, replace, ...])	
break2paragraphs(passage[, return_terminals])	Breaks into paragraphs according to the annotation.
break2sentences(passage[, lang])	Breaks paragraphs into sentences according to the annotation.
contextmanager(func)	@contextmanager decorator.
external_write_mode(*args, **kwargs)	
extract_terminals(p)	returns an iterator of the terminals of the passage p
get_lang(passage_context)	
get_nlp([lang])	Load spaCy model for a given language, determined by 'models' dict or by MODEL_ENV_VAR
get_tokenizer([tokenized, lang])	
get_vocab([vocab, lang])	
get_word_vectors([dim, size, filename, vocab])	Get word vectors from spaCy model or from text file :param dim: dimension to trim vectors to (default: keep original) :param size: maximum number of vectors to load (default: all) :param filename: text file to load vectors from (default: from spaCy model) :param vocab: instead of strings, look up keys of returned dict in vocab (use lang str, e.g.
indent_xml(xml_as_string)	Indents a string of XML-like objects.
is_annotated(passage[, as_array, as_extra])	Whether the passage is already annotated or only partially annotated
load_spacy_model(model)	
read_word_vectors(dim, size, filename)	Read word vectors from text file, with an optional first row indicating size and dimension :param dim: dimension to trim vectors to :param size: maximum number of vectors to load :param filename: text file to load vectors from :return: generator: first element is (#vectors, #dims); and all the rest are (word [string], vector [NumPy array])
set_docs(annotated, as_array, as_extra, ...)	Given spaCy annotations, set values in layer0.extra per paragraph if as_array=True, and in Terminal.extra if as_extra=True

Continued on next page

Table 52 – continued from previous page

to_annotate(passage_contexts, replace[, ...])	Filter passages to get only those that require annotation; split to paragraphs and return generator of (list of tokens, (paragraph index, list of Terminals, Passage) + original context appended) tuples

annotate

ucca.textutil.**annotate**(*passage*, **args*, ***kwargs*)

Run spaCy pipeline on the given passage, unless already annotated :param passage: Passage object, whose layer 0 nodes will be added entries in the 'extra' dict

annotate_all

ucca.textutil.**annotate_all**(*passages*, *replace=False*, *as_array=False*, *as_extra=True*, *as_tuples=False*, *lang='en'*, *vocab=None*, *verbose=False*)

Run spaCy pipeline on the given passages, unless already annotated :param passages: iterable of Passage objects, whose layer 0 nodes will be added entries in the 'extra' dict :param replace: even if a given passage is already annotated, replace with new annotation :param as_array: instead of adding 'extra' entries to each terminal, set layer 0 extra["doc"] to array of ids :param as_extra: set 'extra' entries to each terminal :param as_tuples: treat input as tuples of (passage text, context), and return context for each passage as-is :param lang: optional two-letter language code, will be overridden if passage has "lang" attrib :param vocab: optional dictionary of vocabulary IDs to string values, to avoid loading spaCy model :param verbose: whether to print annotated text :return: generator of annotated passages, which are actually modified in-place (same objects as input)

annotate_as_tuples

ucca.textutil.**annotate_as_tuples**(*passages*, *replace=False*, *as_array=False*, *as_extra=True*, *lang='en'*, *vocab=None*, *verbose=False*)

break2paragraphs

ucca.textutil.**break2paragraphs**(*passage*, *return_terminals=False*, **args*, ***kwargs*)

Breaks into paragraphs according to the annotation.

Uses the 'paragraph' attribute of layer 0 to find paragraphs. :param passage: the Passage object to operate on :param return_terminals: whether to return actual Terminal objects of all terminals rather than just end positions :return: a list of positions in the Passage, each denotes a closing Terminal of a paragraph.

break2sentences

ucca.textutil.**break2sentences**(*passage*, *lang='en'*, **args*, ***kwargs*)

Breaks paragraphs into sentences according to the annotation.

A sentence is a list of terminals which ends with a mark from SENTENCE_END_MARKS, and is also the end of a paragraph or parallel scene. :param passage: the Passage object to operate on :param lang: optional two-letter language code :return: a list of positions in the Passage, each denotes a closing Terminal of a sentence.

external_write_mode

ucca.textutil.**external_write_mode**(**args*, ***kwargs*)

extract_terminals

ucca.textutil.**extract_terminals**(*p*)
returns an iterator of the terminals of the passage p

get_lang

ucca.textutil.**get_lang**(*passage_context*)

get_nlp

ucca.textutil.**get_nlp**(*lang='en'*)
Load spaCy model for a given language, determined by 'models' dict or by MODEL_ENV_VAR

get_tokenizer

ucca.textutil.**get_tokenizer**(*tokenized=False*, *lang='en'*)

get_vocab

ucca.textutil.**get_vocab**(*vocab=None*, *lang=None*)

get_word_vectors

ucca.textutil.**get_word_vectors**(*dim=None*, *size=None*, *filename=None*, *vocab=None*)
Get word vectors from spaCy model or from text file :param dim: dimension to trim vectors to (default: keep original) :param size: maximum number of vectors to load (default: all) :param filename: text file to load vectors from (default: from spaCy model) :param vocab: instead of strings, look up keys of returned dict in vocab (use lang str, e.g. "en", for spaCy vocab) :return: tuple of (dict of word [string or integer] -> vector [NumPy array], dimension)

indent_xml

ucca.textutil.**indent_xml**(*xml_as_string*)
Indents a string of XML-like objects.

This works only for units with no text or tail members, and only for strings whose leaves are written as <tag /> and not <tag></tag>. :param xml_as_string: XML string to indent :return: indented XML string

is_annotated

ucca.textutil.**is_annotated**(*passage*, *as_array=False*, *as_extra=True*)
Whether the passage is already annotated or only partially annotated

load_spacy_model

ucca.textutil.**load_spacy_model**(*model*)

read_word_vectors

ucca.textutil.**read_word_vectors**(*dim*, *size*, *filename*)
Read word vectors from text file, with an optional first row indicating size and dimension :param dim: dimension to trim vectors to :param size: maximum number of vectors to load :param filename: text file to load vectors from :return: generator: first element is (#vectors, #dims); and all the rest are (word [string], vector [NumPy array])

set_docs

ucca.textutil.**set_docs**(*annotated*, *as_array*, *as_extra*, *lang*, *vocab*, *replace*, *verbose*)
Given spaCy annotations, set values in layer0.extra per paragraph if as_array=True, and in Terminal.extra if as_extra=True

to_annotate

ucca.textutil.**to_annotate**(*passage_contexts*, *replace*, *as_array=False*, *as_extra=True*)
Filter passages to get only those that require annotation; split to paragraphs and return generator of (list of tokens, (paragraph index, list of Terminals, Passage) + original context appended) tuples

Classes

Attr	Wrapper for spaCy Attr, determining order for saving in layer0.extra per token when as_array=True
Enum	Generic enumeration.
OrderedDict	Dictionary that remembers insertion order
attrgetter	attrgetter(attr, ...) –> attrgetter object
deque	deque([iterable[, maxlen]]) –> deque object
groupby(iterable[, key])	keys and groups from the iterable.
islice	islice(iterable, stop) –> islice object islice(iterable, start, stop[, step]) –> islice object
itemgetter	itemgetter(item, ...) –> itemgetter object
tqdm([iterable, desc, total, leave, file, ...])	Decorate an iterable object, returning an iterator which acts exactly like the original iterable, but prints a dynamically updating progressbar every time a value is requested.

Attr

class ucca.textutil.**Attr**

Bases: enum.Enum

Wrapper for spaCy Attr, determining order for saving in layer0.extra per token when as_array=True

Attributes Summary

DEP	
ENT_IOB	
ENT_TYPE	
HEAD	
LEMMA	
ORTH	
POS	
PREFIX	
SHAPE	
SUFFIX	
TAG	
key	String used in 'extra' dict of Terminals to store this attribute when as_array=False

Methods Summary

__call__(value[, vocab, as_array, lang])	Resolve numeric ID of attribute value to string (if as_array=False) or to int (if as_array=True)

Attributes Documentation

DEP = 6

ENT_IOB = 5

ENT_TYPE = 4

HEAD = 7

LEMMA = 1

ORTH = 0

POS = 3

PREFIX = 9

SHAPE = 8

SUFFIX = 10

TAG = 2

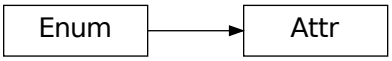
key

String used in 'extra' dict of Terminals to store this attribute when as_array=False

Methods Documentation

__call__(*value*, *vocab=None*, *as_array=False*, *lang=None*)
Resolve numeric ID of attribute value to string (if as_array=False) or to int (if as_array=True)

Class Inheritance Diagram



2.1.12 ucca.validation Module

Functions

join(items)	
tag_to_edge(edges)	
validate(passage[, linkage, multigraph])	

join

ucca.validation.**join**(*items*)

tag_to_edge

ucca.validation.**tag_to_edge**(*edges*)

validate

ucca.validation.**validate**(*passage*, *linkage=True*, *multigraph=False*)

Classes

ETags	alias of *ucca.layer1.EdgeTags*
L0Tags	alias of *ucca.layer0.NodeTags*
L1Tags	alias of *ucca.layer1.NodeTags*
NodeValidator(node)	
attrgetter	attrgetter(attr, ...) –> attrgetter object
groupby(iterable[, key])	keys and groups from the iterable.

NodeValidator

class ucca.validation.**NodeValidator**(*node*)

Bases: object

Methods Summary

validate_foundational()	
validate_linkage()	
validate_non_terminal([linkage, multigraph])	
validate_terminal()	
validate_top_level()	

Methods Documentation

validate_foundational()

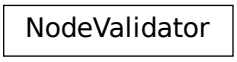
validate_linkage()

validate_non_terminal(*linkage=False*, *multigraph=False*)

validate_terminal()

validate_top_level()

Class Inheritance Diagram



2.1.13 ucca.visualization Module

Functions

draw(passage[, node_ids])	
node_label(node)	
standoff(p)	Visualize to Standoff .ann format, which can be presented with brat :param p: Passage :return: string in Standoff format
tex_escape(text)	**param text** a plain text message

Continued on next page

Table 59 – continued from previous page

tikz(p[, indent, node_ids])	Visualize to TikZ format :param p: Passage :param indent: indentation size or None for no indentation :param node_ids: whether to include node IDs :return: string in TikZ format
topological_layout(passage)	

draw

ucca.visualization.**draw**(*passage*, *node_ids=False*)

node_label

ucca.visualization.**node_label**(*node*)

standoff

ucca.visualization.**standoff**(*p*)

Visualize to Standoff .ann format, which can be presented with brat :param p: Passage :return: string in Standoff format

tex_escape

ucca.visualization.**tex_escape**(*text*)

Parameters **text** – a plain text message

Returns the message escaped to appear correctly in LaTeX

tikz

ucca.visualization.**tikz**(*p*, *indent=None*, *node_ids=False*)

Visualize to TikZ format :param p: Passage :param indent: indentation size or None for no indentation :param node_ids: whether to include node IDs :return: string in TikZ format

topological_layout

ucca.visualization.**topological_layout**(*passage*)

2.2 Scripts Documentation

2.2.1 scripts.annotate Module

Functions

annotate_all(passages[, replace, as_array, ...])	Run spaCy pipeline on the given passages, unless already annotated :param passages: iterable of Passage objects, whose layer 0 nodes will be added entries in the 'extra' dict :param replace: even if a given passage is already annotated, replace with new annotation :param as_array: instead of adding 'extra' entries to each terminal, set layer 0 extra["doc"] to array of ids :param as_extra: set 'extra' entries to each terminal :param as_tuples: treat input as tuples of (passage text, context), and return context for each passage as-is :param lang: optional two-letter language code, will be overridden if passage has "lang" attrib :param vocab: optional dictionary of vocabulary IDs to string values, to avoid loading spaCy model :param verbose: whether to print annotated text :return: generator of annotated passages, which are actually modified in-place (same objects as input)
get_passages_with_progress_bar(filename_patterns)	
is_annotated(passage[, as_array, as_extra])	Whether the passage is already annotated or only partially annotated
main(args)	
write_passage(passage[, output_format, ...])	Write a given UCCA passage in any format.

main

scripts.annotate.**main**(*args*)

2.2.2 scripts.convert_1_0_to_1_2 Module

Functions

annotate_all(passages[, replace, as_array, ...])	Run spaCy pipeline on the given passages, unless already annotated :param passages: iterable of Passage objects, whose layer 0 nodes will be added entries in the 'extra' dict :param replace: even if a given passage is already annotated, replace with new annotation :param as_array: instead of adding 'extra' entries to each terminal, set layer 0 extra["doc"] to array of ids :param as_extra: set 'extra' entries to each terminal :param as_tuples: treat input as tuples of (passage text, context), and return context for each passage as-is :param lang: optional two-letter language code, will be overridden if passage has "lang" attrib :param vocab: optional dictionary of vocabulary IDs to string values, to avoid loading spaCy model :param verbose: whether to print annotated text :return: generator of annotated passages, which are actually modified in-place (same objects as input)
convert_passage(passage, report_writer)	
copy_edge(edge[, parent, child, tag, attrib])	

Continued on next page

Table 61 – continued from previous page

destroy(node_or_edge)	
extract_aux(terminal, parent, grandparent)	
extract_ground(terminal, parent, grandparent)	
extract_modal(terminal, parent, grandparent)	
extract_relator(terminal, parent, grandparent)	
extract_that(terminal, parent, grandparent)	
fix_punct(terminal, parent, grandparent)	
fix_root_terminal_child(terminal, parent, ...)	
fix_unary_participant(terminal, parent, ...)	
flag_relator_starts_main_relation(terminal, ...)	
flag_suspected_secondary(terminal, parent, ...)	
fparent(node_or_edge)	
get_annotation(terminal, attr)	
get_passages_with_progress_bar(filename_patterns)	
is_main_relation(node)	
main(args)	
move_node(node, new_parent[, tag])	
remove(parent, child)	
set_light_verb_function(terminal, parent, ...)	
write_passage(passage[, output_format, ...])	Write a given UCCA passage in any format.

convert_passage

scripts.convert_1_0_to_1_2.**convert_passage**(*passage*, *report_writer*)

extract_aux

scripts.convert_1_0_to_1_2.**extract_aux**(*terminal*, *parent*, *grandparent*)

extract_ground

scripts.convert_1_0_to_1_2.**extract_ground**(*terminal*, *parent*, *grandparent*)

extract_modal

scripts.convert_1_0_to_1_2.**extract_modal**(*terminal*, *parent*, *grandparent*)

extract_relator

scripts.convert_1_0_to_1_2.**extract_relator**(*terminal*, *parent*, *grandparent*)

extract_that

scripts.convert_1_0_to_1_2.**extract_that**(*terminal*, *parent*, *grandparent*)

fix_punct

scripts.convert_1_0_to_1_2.**fix_punct**(*terminal*, *parent*, *grandparent*)

fix_root_terminal_child

scripts.convert_1_0_to_1_2.**fix_root_terminal_child**(*terminal*, *parent*, *grandparent*)

fix_unary_participant

scripts.convert_1_0_to_1_2.**fix_unary_participant**(*terminal*, *parent*, *grandparent*)

flag_relator_starts_main_relation

scripts.convert_1_0_to_1_2.**flag_relator_starts_main_relation**(*terminal*, *parent*, *grandparent*)

flag_suspected_secondary

scripts.convert_1_0_to_1_2.**flag_suspected_secondary**(*terminal*, *parent*, *grandparent*)

get_annotation

scripts.convert_1_0_to_1_2.**get_annotation**(*terminal*, *attr*)

is_main_relation

scripts.convert_1_0_to_1_2.**is_main_relation**(*node*)

main

scripts.convert_1_0_to_1_2.**main**(*args*)

move_node

scripts.convert_1_0_to_1_2.**move_node**(*node*, *new_parent*, *tag=None*)

set_light_verb_function

scripts.convert_1_0_to_1_2.**set_light_verb_function**(*terminal*, *parent*, *grandparent*)

2.2.3 scripts.convert_2_0_to_1_2 Module

Functions

convert_passage(passage, report_writer)	
copy_edge(edge[, parent, child, tag, attrib])	
destroy(node_or_edge)	
get_passages_with_progress_bar(filename_patterns)	
main(args)	
replace_time_and_quantifier(edge)	
write_passage(passage[, output_format, ...])	Write a given UCCA passage in any format.

convert_passage

scripts.convert_2_0_to_1_2.**convert_passage**(*passage*, *report_writer*)

main

scripts.convert_2_0_to_1_2.**main**(*args*)

replace_time_and_quantifier

scripts.convert_2_0_to_1_2.**replace_time_and_quantifier**(*edge*)

2.2.4 scripts.count_parents_children Module

Functions

clip(l, m)	
get_passages_with_progress_bar(filename_patterns)	
main(args)	
plot_histogram(counter, label[, plot])	
plot_pie(counter, label[, plot])	

clip

scripts.count_parents_children.**clip**(*l*, *m*)

main

scripts.count_parents_children.**main**(*args*)

plot_histogram

scripts.count_parents_children.**plot_histogram**(*counter*, *label*, *plot=None*)

plot_pie

scripts.count_parents_children.**plot_pie**(*counter*, *label*, *plot=None*)

2.2.5 scripts.evaluate_db Module

The evaluation software for UCCA layer 1.

Functions

evaluate(guessed, ref[, converter, verbose, ...])	Compare two passages and return requested diagnostics and scores, possibly printing them too.
main(args)	

main

scripts.evaluate_db.**main**(*args*)

2.2.6 scripts.evaluate_standard Module

The evaluation script for UCCA layer 1.

Functions

check_args(args)	
main(args)	
match_by_id(guessed, ref)	
print_f1(result, eval_type)	
summarize(args, results, eval_type)	

check_args

scripts.evaluate_standard.**check_args**(*args*)

main

scripts.evaluate_standard.**main**(*args*)

match_by_id

scripts.evaluate_standard.**match_by_id**(*guessed*, *ref*)

print_f1

scripts.evaluate_standard.**print_f1**(*result*, *eval_type*)

summarize

scripts.evaluate_standard.**summarize**(*args*, *results*, *eval_type*)

2.2.7 scripts.find_constructions Module

Functions

add_argument(argparser[, default])	
external_write_mode(*args, **kwargs)	
extract_candidates(passage[, constructions, ...])	Find candidate edges by constructions in UCCA passage.
get_passages_with_progress_bar(filename_patterns)	
main(args)	

main

scripts.find_constructions.**main**(*args*)

2.2.8 scripts.fix_tokenization Module

Functions

context(i, terminals)	
create_token_element(state, text, is_punctuation)	
create_unit_element(state, text, tag)	
decode_special_chars(tokens)	
expand_to_neighboring_punct(i, is_puncts)	>>> expand_to_neighboring_punct(0, ↪[False, True, True])
false_indices(l)	
fix_tokenization(passage, words_set, lang, cw)	
from_site(elem)	Converts site XML structure to core.Passage object.
get_parents(paragraph, elements)	
get_passages_with_progress_bar(filename_patterns)	
get_tokenizer([tokenized, lang])	
handle_words_set(rule, i, terminals, ...)	use set of words to determine the right fix needed
insert_punct(insert_index, ...)	
insert_retokenized(terminal, ...)	
insert_retokenized_currency(i, terminals, ...)	
insert_spaces(tokens)	
is_punct(text)	
main(args)	
normalize(passage[, extra])	
read_dict(file)	

Continued on next page

Table 67 – continued from previous page

retokenize(i, start, end, terminals, ...)	
split_apostrophe_to_units(i, terminals, ...)	split token with apostrophe to Elaborator and Center.
split_apostrophe_unanalyzable(i, terminals, ...)	Split apostrophe as unanalyzable.
split_hyphen_to_units(i, terminals, ...)	split token with hyphen to two different units.
split_hyphen_unanalyzable(i, terminals, ...)	split token with hyphens to unanalyzable tokens.
split_possessive_s_to_units(i, terminals, ...)	split possessive s to two different units.
split_possessive_s_unanalyzable(i, ...)	split possessive s as unanalyzable.
strip_context(new_context, old_context, ...)	`>>> strip_context(["I", "'ve", "done"], [ ↪"I", "'ve", "done"], 1, 1)`
to_site(passage)	Converts a passage to the site XML format.
write_passage(passage[, output_format, ...])	Write a given UCCA passage in any format.

context

scripts.fix_tokenization.**context**(*i*, *terminals*)

create_token_element

scripts.fix_tokenization.**create_token_element**(*state*, *text*, *is_punctuation*)

create_unit_element

scripts.fix_tokenization.**create_unit_element**(*state*, *text*, *tag*)

decode_special_chars

scripts.fix_tokenization.**decode_special_chars**(*tokens*)

expand_to_neighboring_punct

scripts.fix_tokenization.**expand_to_neighboring_punct**(*i*, *is_puncts*)

```
>>> expand_to_neighboring_punct(0, [False, True, True])
(0, 3)
>>> expand_to_neighboring_punct(2, [True, True, False])
(0, 3)
>>> expand_to_neighboring_punct(1, [False, False, False])
(1, 2)
```

false_indices

scripts.fix_tokenization.**false_indices**(*l*)

fix_tokenization

scripts.fix_tokenization.**fix_tokenization**(*passage*, *words_set*, *lang*, *cw*)

get_parents

scripts.fix_tokenization.**get_parents**(*paragraph*, *elements*)

handle_words_set

scripts.fix_tokenization.**handle_words_set**(*rule*, *i*, *terminals*, *preterminals*, *preterminal_parents*, *state*)

use set of words to determine the right fix needed

insert_punct

scripts.fix_tokenization.**insert_punct**(*insert_index*, *preterminal_parent*, *state*, *punct_tokens*)

insert_retokenized

scripts.fix_tokenization.**insert_retokenized**(*terminal*, *preterminal_parent*, *tokens*, *index_in_preterminal_parent*, *non_punct_index*, *state*)

insert_retokenized_currency

scripts.fix_tokenization.**insert_retokenized_currency**(*i*, *terminals*, *preterminals*, *preterminal_parents*, *tokens*, *state*)

insert_spaces

scripts.fix_tokenization.**insert_spaces**(*tokens*)

is_punct

scripts.fix_tokenization.**is_punct**(*text*)

main

scripts.fix_tokenization.**main**(*args*)

read_dict

scripts.fix_tokenization.**read_dict**(*file*)

retokenize

scripts.fix_tokenization.**retokenize**(*i*, *start*, *end*, *terminals*, *preterminals*, *preterminal_parents*, *passage_id*, *tokenizer*, *state*, *cw*, *words*)

split_apostrophe_to_units

scripts.fix_tokenization.**split_apostrophe_to_units**(*i*, *terminals*, *preterminals*, *preterminal_parents*, *tag1*, *tag2*, *state*)

split token with apostrophe to Elaborator and Center. x'xxx -> x' xxx

split_apostrophe_unanalyzable

scripts.fix_tokenization.**split_apostrophe_unanalyzable**(*i*, *terminals*, *preterminals*, *preterminal_parents*, *state*)

Split apostrophe as unanalyzable. x'xxx -> x' xxx. use when the original token is unanalyzable.

split_hyphen_to_units

scripts.fix_tokenization.**split_hyphen_to_units**(*i*, *terminals*, *preterminals*, *preterminal_parents*, *tag1*, *tag2*, *state*)

split token with hyphen to two different units. xxx-xxx -> xxx - xxx

split_hyphen_unanalyzable

scripts.fix_tokenization.**split_hyphen_unanalyzable**(*i*, *terminals*, *preterminals*, *preterminal_parents*, *state*)

split token with hyphens to unanalyzable tokens. xxx-xxx-xx -> xxx - xxx - xx

split_possessive_s_to_units

scripts.fix_tokenization.**split_possessive_s_to_units**(*i*, *terminals*, *preterminals*, *preterminal_parents*, *state*, *tag1*, *tag2*)

split possessive s to two different units. xxx's -> xxx 's

split_possessive_s_unanalyzable

scripts.fix_tokenization.**split_possessive_s_unanalyzable**(*i*, *terminals*, *preterminals*, *preterminal_parents*, *state*)

split possessive s as unanalyzable. xxx's -> xxx 's. use when the original token is unanalyzable.

strip_context

scripts.fix_tokenization.**strip_context**(*new_context*, *old_context*, *start_offset*, *end_offset*)

```
>>> strip_context(["I", "'ve", "done"], ["I", "'ve", "done"], 1, 1)
["'ve"]
>>> strip_context(["I", "'ve,", "done"], ["I", "'ve", ",", "done"], 1, 1)
["'ve", ","]
>>> strip_context(["'ve", "done"], ["'", "ve", "done"], 0, 1)
["'", "ve"]
>>> strip_context(["I", "'ve,"], ["I", "'ve", ","], 1, 0)
["'ve", ","]
>>> strip_context(["can", "'t", "see"], ["ca", "n't", "see"], 1, 1)
["'t"]
>>> strip_context(["I", "can", "'t"], ["I", "ca", "n't"], 1, 1)
["can"]
>>> strip_context(["because", "somebody", "'d"], ["because", "somebody'd"], 1, 1)
[]
>>> strip_context(["somebody", "'d", "always"], ["somebody'd", "always"], 1, 1)
[]
```

Classes

Element	
SiteCfg	Contains static configuration for conversion to/from the site XML.
SiteUtil	Contains utility functions for converting to/from the site XML.
State()	

State

class scripts.fix_tokenization.**State**

Bases: object

Methods Summary

get_id()	

Methods Documentation

get_id()

Class Inheritance Diagram

2.2.9 scripts.join_passages Module

Functions

get_passages(filename_patterns, **kwargs)	
main(args)	
passage2file(passage, filename[, indent, binary])	Writes a UCCA passage as a standard XML file or a binary pickle :param passage: passage object to write :param filename: file name to write to :param indent: whether to indent each line :param binary: whether to write pickle format (or XML)

main

scripts.join_passages.**main**(*args*)

2.2.10 scripts.join_sdp Module

Functions

main(args)	

main

scripts.join_sdp.**main**(*args*)

2.2.11 scripts.load_word_vectors Module

Functions

get_word_vectors([dim, size, filename, vocab])	Get word vectors from spaCy model or from text file :param dim: dimension to trim vectors to (default: keep original) :param size: maximum number of vectors to load (default: all) :param filename: text file to load vectors from (default: from spaCy model) :param vocab: instead of strings, look up keys of returned dict in vocab (use lang str, e.g.
main(args)	

main

scripts.load_word_vectors.**main**(*args*)

2.2.12 scripts.normalize Module

Functions

get_passages_with_progress_bar(filename_patterns)	
main(args)	
normalize(passage[, extra])	
write_passage(passage[, output_format, ...])	Write a given UCCA passage in any format.

main

scripts.normalize.**main**(*args*)

2.2.13 scripts.pickle_to_standard Module

Functions

file2passage(filename)	Opens a file and returns its parsed Passage object Tries to read both as a standard XML file and as a binary pickle :param filename: file name to write to
main(args)	
passage2file(passage, filename[, indent, binary])	Writes a UCCA passage as a standard XML file or a binary pickle :param passage: passage object to write :param filename: file name to write to :param indent: whether to indent each line :param binary: whether to write pickle format (or XML)

main

scripts.pickle_to_standard.**main**(*args*)

2.2.14 scripts.replace_tokens_by_dict Module

Functions

glob(pathname, *[, recursive])	Return a list of paths matching a pathname pattern.
main(args)	
read_dictionary_from_file(filename)	

main

scripts.replace_tokens_by_dict.**main**(*args*)

read_dictionary_from_file

scripts.replace_tokens_by_dict.**read_dictionary_from_file**(*filename*)

2.2.15 scripts.site_pickle_to_standard Module

Functions

glob(pathname, *[, recursive])	Return a list of paths matching a pathname pattern.
main(args)	
pickle_site2passage(filename)	Opens a pickle file containing XML in UCCA site format and returns its parsed Passage object
write_passage(passage[, output_format, ...])	Write a given UCCA passage in any format.

main

scripts.site_pickle_to_standard.**main**(*args*)

pickle_site2passage

scripts.site_pickle_to_standard.**pickle_site2passage**(*filename*)
Opens a pickle file containing XML in UCCA site format and returns its parsed Passage object

2.2.16 scripts.site_to_standard Module

Functions

check_illegal_combinations(args)	
db2passage(handle, pid, user)	Gets the annotation of user to pid from the DB handle - returns a passage
fromstring(text[, parser])	Parse XML document from string constant.
glob(pathname, *[, recursive])	Return a list of paths matching a pathname pattern.
main(args)	
site2passage(filename)	Opens a file and returns its parsed Passage object
write_passage(passage[, output_format, ...])	Write a given UCCA passage in any format.

check_illegal_combinations

scripts.site_to_standard.**check_illegal_combinations**(*args*)

db2passage

scripts.site_to_standard.**db2passage**(*handle*, *pid*, *user*)
Gets the annotation of user to pid from the DB handle - returns a passage

main

scripts.site_to_standard.**main**(*args*)

site2passage

scripts.site_to_standard.**site2passage**(*filename*)
Opens a file and returns its parsed Passage object

2.2.17 scripts.site_to_text Module

Functions

db2passage(handle, pid, user)	Gets the annotation of user to pid from the DB handle - returns a passage
fromstring(text[, parser])	Parse XML document from string constant.
main(args)	
site2passage(filename)	Opens a file and returns its parsed Passage object

db2passage

scripts.site_to_text.**db2passage**(*handle*, *pid*, *user*)
Gets the annotation of user to pid from the DB handle - returns a passage

main

scripts.site_to_text.**main**(*args*)

site2passage

scripts.site_to_text.**site2passage**(*filename*)
Opens a file and returns its parsed Passage object

2.2.18 scripts.split_corpus Module

Functions

copy(src, dest[, link])	
copyfile(src, dst, *[, follow_symlinks])	Copy data from src to dst.
main(args)	
not_split_dir(filename)	
numeric(s)	
split_passages(directory, train, dev, link)	

copy

scripts.split_corpus.**copy**(*src*, *dest*, *link=False*)

main

scripts.split_corpus.**main**(*args*)

not_split_dir

scripts.split_corpus.**not_split_dir**(*filename*)

numeric

scripts.split_corpus.**numeric**(*s*)

split_passages

scripts.split_corpus.**split_passages**(*directory*, *train*, *dev*, *link*, *quiet=False*)

2.2.19 scripts.standard_to_pickle Module

Functions

external_write_mode(*args, **kwargs)	
file2passage(filename)	Opens a file and returns its parsed Passage object Tries to read both as a standard XML file and as a binary pickle :param filename: file name to write to
main(args)	
passage2file(passage, filename[, indent, binary])	Writes a UCCA passage as a standard XML file or a binary pickle :param passage: passage object to write :param filename: file name to write to :param indent: whether to indent each line :param binary: whether to write pickle format (or XML)

main

scripts.standard_to_pickle.**main**(*args*)

2.2.20 scripts.standard_to_sentences Module

Functions

external_write_mode(*args, **kwargs)	
extract_terminals(p)	returns an iterator of the terminals of the passage p
get_passages_with_progress_bar(filename_patterns)	
main(args)	
normalize(passage[, extra])	
passage2file(passage, filename[, indent, binary])	Writes a UCCA passage as a standard XML file or a binary pickle :param passage: passage object to write :param filename: file name to write to :param indent: whether to indent each line :param binary: whether to write pickle format (or XML)
split2sentences(passage[, remarks, lang, ids])	
split_passage(passage, ends[, remarks, ids, ...])	Split the passage on the given terminal positions :param passage: passage to split :param ends: sequence of positions at which the split passages will end :param remarks: add original node ID as remarks to the new nodes :param ids: optional iterable of ids, the same length as ends, to set passage IDs for each split :param suffix_format: in case ids is None, use this format for the running index suffix :param suffix_start: in case ids is None, use this starting index for the running index suffix :return: sequence of passages
warning(msg, *args, **kwargs)	Log a message with severity 'WARNING' on the root logger.

main

scripts.standard_to_sentences.**main**(*args*)

Classes

Splitter(sentences[, enum, suffix_format, ...])	
count	count(start=0, step=1) –> count object

Splitter

class scripts.standard_to_sentences.**Splitter**(*sentences*, *enum=False*, *suffix_format='%03d'*, *suffix_start=0*)

Bases: object

Methods Summary

read_file(filename, **kwargs)	
split(passage)	

Methods Documentation

classmethod read_file (*filename*, ***kwargs*)

split (*passage*)

Class Inheritance Diagram

2.2.21 scripts.standard_to_site Module

Functions

external_write_mode(*args, **kwargs)	
get_passages_with_progress_bar(filename_patterns)	
main(args)	
tostring(element[, encoding, method, ...])	Generate string representation of XML element.

main

scripts.standard_to_site.**main** (*args*)

2.2.22 scripts.standard_to_text Module

Functions

file2passage(filename)	Opens a file and returns its parsed Passage object Tries to read both as a standard XML file and as a binary pickle :param filename: file name to write to
get_passages_with_progress_bar(filename_patterns)	
glob(pathname, *[, recursive])	Return a list of paths matching a pathname pattern.
main(args)	
numeric(x)	

Continued on next page

Table 85 – continued from previous page

to_text(passage[, sentences, lang])	Converts from a Passage object to tokenized strings.
write_text(passage, f, sentences, lang[, ...])	

main

scripts.standard_to_text.**main**(*args*)

numeric

scripts.standard_to_text.**numeric**(*x*)

write_text

scripts.standard_to_text.**write_text**(*passage*, *f*, *sentences*, *lang*, *prepend_id=False*)

2.2.23 scripts.statistics Module

Functions

get_passages_with_progress_bar(filename_patterns)	
main(args)	

main

scripts.statistics.**main**(*args*)

2.2.24 scripts.unique_roles Module

Functions

get_passages_with_progress_bar(filename_patterns)	
main(args)	

main

scripts.unique_roles.**main**(*args*)

2.2.25 scripts.validate Module

Functions

Pool	Returns a process pool object
check_args(parser, args)	

Continued on next page

Table 88 – continued from previous page

external_write_mode(*args, **kwargs)
get_passages_with_progress_bar(filename_patterns)
main(args)
normalize(passage[, extra])
print_errors(passage_id, errors[, id_len])
validate(passage[, linkage, multigraph])

check_args

scripts.validate.**check_args**(*parser*, *args*)

main

scripts.validate.**main**(*args*)

print_errors

scripts.validate.**print_errors**(*passage_id*, *errors*, *id_len=None*)

Classes

Validator([normalization, extra, linkage, ...])

Validator

class scripts.validate.**Validator**(*normalization=False*, *extra=False*, *linkage=True*, *multigraph=False*, *strict=False*)

Bases: object

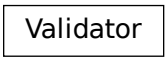
Methods Summary

validate_passage(passage)

Methods Documentation

validate_passage(*passage*)

Class Inheritance Diagram



2.2.26 scripts.visualize Module

Functions

external_write_mode(*args, **kwargs)	
get_passages(filename_patterns, **kwargs)	
get_passages_with_progress_bar(filename_patterns)	
main(args)	
print_text(args, text, suffix)	
split2sentences(passage[, remarks, lang, ids])	

main

scripts.visualize.**main**(*args*)

print_text

scripts.visualize.**print_text**(*args*, *text*, *suffix*)

2.3 UCCA DB Documentation

2.3.1 ucca_db.api Module

Functions

external_write_mode(*args, **kwargs)	
fromstring(text)	
fromstring_xml(text[, parser])	Parse XML document from string constant.
get_by_xids(host_name, db_name, xids, **kwargs)	Returns the passages that correspond to xids (which is a list of them)
get_connection(db_name, host_name)	connects to the db and host, returns a connection object
get_cursor(host_name, db_name)	create a cursor to the search path
get_most_recent_passage_by_uid(uid, ...[, ...])	

Continued on next page

Table 92 – continued from previous page

get_most_recent_xids(host_name, db_name, ...)	Returns the most recent xids of the given username.
get_passage(host_name, db_name, pid)	Returns the passages with the given id numbers
get_predicates(host_name, db_name[, ...])	Returns a list of all the predicates in the UCCA corpus.
get_uid(host_name, db_name, username)	Returns the uid matching the given username.
get_xml_trees(host_name, db_name, pid[, ...])	Params: db, host, paragraph id, the list of usernames wanted, Optional: graceful: True if no excpetions are to be raised excpetion raised if a user did not submit an annotation for the passage returns a list of xml roots elements
get_xmls_by_username(host_name, db_name, ...)	
linkage_type(u)	Returns the type of the primary linkage the scene participates in.
main(argv)	
print_passages_to_file(host_name, db_name, paids)	Returns for that user a list of submitted passages and a list of assigned but not submitted passages.
tostring(element[, encoding, method, ...])	Generate string representation of XML element.
unit_length(u)	Returns the number of terminals (excluding remote units and punctuations) that are descendants of the unit u.
write_to_db(host_name, db_name, xml, ...[, ...])	

fromstring

ucca_db.api.**fromstring**(*text*)

get_by_xids

ucca_db.api.**get_by_xids**(*host_name*, *db_name*, *xids*, ***kwargs*)

Returns the passages that correspond to xids (which is a list of them)

get_connection

ucca_db.api.**get_connection**(*db_name*, *host_name*)

connects to the db and host, returns a connection object

get_cursor

ucca_db.api.**get_cursor**(*host_name*, *db_name*)

create a cursor to the search path

get_most_recent_passage_by_uid

ucca_db.api.**get_most_recent_passage_by_uid**(*uid*, *passage_id*, *host_name*, *db_name*, *verbose=False*, *write_xids=None*, *strict=False*, ***kwargs*)

get_most_recent_xids

ucca_db.api.**get_most_recent_xids**(*host_name*, *db_name*, *username*)
Returns the most recent xids of the given username.

get_passage

ucca_db.api.**get_passage**(*host_name*, *db_name*, *pid*)
Returns the passages with the given id numbers

get_predicates

ucca_db.api.**get_predicates**(*host_name*, *db_name*, *only_complex=True*)
Returns a list of all the predicates in the UCCA corpus. usernames – the names of the users whose completed passages we should take. only_complex – only the multi-word predicates will be returned. start_index – the minimal passage number to be taken into account.

get_uid

ucca_db.api.**get_uid**(*host_name*, *db_name*, *username*)
Returns the uid matching the given username.

get_xml_trees

ucca_db.api.**get_xml_trees**(*host_name*, *db_name*, *pid*, *usernames=None*, *graceful=False*)
Params: db, host, paragraph id, the list of usernames wanted, Optional: graceful: True if no excpetions are to be raised excpetion raised if a user did not submit an annotation for the passage returns a list of xml roots elements

get_xmls_by_username

ucca_db.api.**get_xmls_by_username**(*host_name*, *db_name*, *username*)

linkage_type

ucca_db.api.**linkage_type**(*u*)
Returns the type of the primary linkage the scene participates in. It can be A,E or H. if it is a C, it returns the taf of the first fparent which is an A,E or H. If it does not find an fparent with either of these categories, it returns UNK_LINKAGE_TYPE.

main

ucca_db.api.**main**(*argv*)

print_passages_to_file

ucca_db.api.**print_passages_to_file**(*host_name*, *db_name*, *paids*, *write_xml=False*, *write_site_xml=False*, *prefix=''*, *start_index=0*)

Returns for that user a list of submitted passages and a list of assigned but not submitted passages. Each passage is given in the format: (<passage ID>, <source>, <recent submitted xid or -1 if not submitted>, <number of tokens in the passage>, <number of units in the passage>, <number of scenes in the passage>, <average length of a scene>). It also returns a distribution of the categories. write_xml: determines whether to write it to a file, named <prefix><the number of the xml>.xml skip_first: the index of the passage where it should start looking (the ones before are skipped)

unit_length

ucca_db.api.**unit_length**(*u*)

Returns the number of terminals (excluding remote units and punctuations) that are descendants of the unit u.

write_to_db

ucca_db.api.**write_to_db**(*host_name*, *db_name*, *xml*, *new_pid*, *new_prid*, *username*, *status=1*)

2.3.2 ucca_db.download Module

Functions

external_write_mode(*args, **kwargs)	
get_by_method(method, id_field[, passage_id])	
get_by_xids(host_name, db_name, xids, **kwargs)	Returns the passages that correspond to xids (which is a list of them)
get_most_recent_passage_by_uid(uid, ...[, ...])	
main(args)	
tostring(element[, encoding, method, ...])	Generate string representation of XML element.
write_passage(passage[, output_format, ...])	Write a given UCCA passage in any format.

get_by_method

ucca_db.download.**get_by_method**(*method*, *id_field*, *passage_id=None*, ***kwargs*)

main

ucca_db.download.**main**(*args*)

2.3.3 ucca_db.upload Module

Functions

get_passages_with_progress_bar(filename_patterns)	
main(args)	
tostring(element[, encoding, method, ...])	Generate string representation of XML element.
upload_passage(xml_root[, site_filename, ...])	
write_to_db(host_name, db_name, xml, ...[, ...])	

main

ucca_db.upload.**main**(*args*)

upload_passage

ucca_db.upload.**upload_passage**(*xml_root*, *site_filename=None*, *verbose=False*, ***kwargs*)

2.4 UCCA-App API Documentation

2.4.1 uccaapp.api Module

Classes

ServerAccessor(server_address, email, password)	

ServerAccessor

class uccaapp.api.**ServerAccessor**(*server_address*, *email*, *password*, *auth_token=None*, *verbose=False*, ***kwargs*)

Bases: object

Methods Summary

add_arguments(argparser)	
add_project_id_argument(argparser)	
add_source_id_argument(argparser)	
add_user_id_argument(argparser)	
create(data, prefix)	
create_category(**kwargs)	
create_passage(**kwargs)	
create_task(**kwargs)	
get(_id, prefix)	
get_category(category_id)	
get_layer(layer_id)	
get_passage(passage_id)	
get_project(project_id)	
get_source(source_id)	
get_task(task_id)	

Continued on next page

Table 96 – continued from previous page

get_user(user_id)
get_user_task(task_id)
login(email, password)
request(method, url_suffix, **kwargs)
set_project([project_id])
set_source([source_id])
set_user([user_id])
submit_task([submit])
type(data)
update(data, prefix)
update_passage(**kwargs)
update_task(**kwargs)

Methods Documentation

static add_arguments(*argparser*)

static add_project_id_argument(*argparser*)

static add_source_id_argument(*argparser*)

static add_user_id_argument(*argparser*)

create(*data*, *prefix*)

create_category(***kwargs*)

create_passage(***kwargs*)

create_task(***kwargs*)

get(*_id*, *prefix*)

get_category(*category_id*)

get_layer(*layer_id*)

get_passage(*passage_id*)

get_project(*project_id*)

get_source(*source_id*)

get_task(*task_id*)

get_user(*user_id*)

get_user_task(*task_id*)

login(*email*, *password*)

request(*method*, *url_suffix*, ***kwargs*)

set_project(*project_id=None*)

set_source(*source_id=None*)

set_user(*user_id=None*)

submit_task(*submit=True*, ***kwargs*)

static type(*data*)

update(*data*, *prefix*)

update_passage(***kwargs*)

update_task(***kwargs*)

Class Inheritance Diagram

2.4.2 uccaapp.convert_and_evaluate Module

Functions

evaluate(guessed, ref[, converter, verbose, ...])	Compare two passages and return requested diagnostics and scores, possibly printing them too.
glob(pathname, *[, recursive])	Return a list of paths matching a pathname pattern.
main(filenames, write, **kwargs)	
read_files_and_dirs(files_and_dirs[, ...])	**param files_and_dirs** iterable of files and/or directories to look in

main

uccaapp.convert_and_evaluate.**main**(*filenames*, *write*, ***kwargs*)

2.4.3 uccaapp.copy_categories Module

Functions

add_arguments(argparser)
main(args)

add_arguments

uccaapp.copy_categories.**add_arguments**(*argparser*)

main

uccaapp.copy_categories.**main**(*args*)

2.4.4 uccaapp.create_annotation_tasks Module

Functions

main(**kwargs)	

main

uccaapp.create_annotation_tasks.**main**(***kwargs*)

Classes

AnnotationTaskCreator([project_id])	
ServerAccessor(server_address, email, password)	
tqdm([iterable, desc, total, leave, file, ...])	Decorate an iterable object, returning an iterator which acts exactly like the original iterable, but prints a dynamically updating progressbar every time a value is requested.

AnnotationTaskCreator

class uccaapp.create_annotation_tasks.**AnnotationTaskCreator**(*project_id=None*, ***kwargs*)

Bases: *uccaapp.api.ServerAccessor*

Methods Summary

add_arguments(argparser)	
build_task(user_id, task_id[, review, ...])	
create_tasks(filename[, log])	
read_lines(filename)	

Methods Documentation

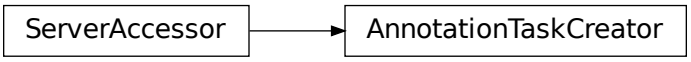
static add_arguments(*argparser*)

build_task(*user_id*, *task_id*, *review=False*, *manager_comment=None*, *strict=False*, ***kwargs*)

create_tasks(*filename*, *log=None*, ***kwargs*)

static read_lines(*filename*)

Class Inheritance Diagram



2.4.5 uccaapp.create_tokenization_tasks Module

Functions

main(**kwargs)

main

uccaapp.create_tokenization_tasks.**main**(***kwargs*)

Classes

AnnotationTaskCreator([project_id])
ServerAccessor(server_address, email, password)
TokenizationTaskCreator(project_id, **kwargs)

TokenizationTaskCreator

class uccaapp.create_tokenization_tasks.**TokenizationTaskCreator**(*project_id*, ***kwargs*)

Bases: *uccaapp.create_annotation_tasks.AnnotationTaskCreator*

Methods Summary

add_arguments(argparser)
build_task(user_id, passage_id, **kwargs)

Methods Documentation

static add_arguments(*argparser*)

build_task(*user_id*, *passage_id*, ***kwargs*)

Class Inheritance Diagram

2.4.6 uccaapp.download_task Module

Functions

from_json(lines, *args[, ...])	Convert text (or dict) in UCCA-App JSON format to a Passage object.
main(**kwargs)	
write_passage(passage[, output_format, ...])	Write a given UCCA passage in any format.

main

uccaapp.download_task.**main**(***kwargs*)

Classes

ServerAccessor(server_address, email, password)	
TaskDownloader(**kwargs)	
tqdm([iterable, desc, total, leave, file, ...])	Decorate an iterable object, returning an iterator which acts exactly like the original iterable, but prints a dynamically updating progressbar every time a value is requested.

TaskDownloader

class uccaapp.download_task.**TaskDownloader**(***kwargs*)

Bases: *uccaapp.api.ServerAccessor*

Methods Summary

add_arguments(argparser)	
add_write_arguments(argparser)	
download_task(task_id[, normalize, write, ...])	
download_tasks(task_ids[, by_filename, ...])	

Methods Documentation

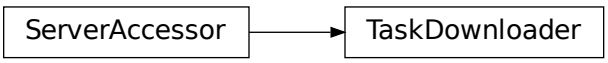
static add_arguments(*argparser*)

static add_write_arguments(*argparser*)

download_task(*task_id*, *normalize=False*, *write=True*, *validate=None*, *binary=None*, *log=None*, *out_dir=None*, *prefix=None*, *by_external_id=False*, *verbose=False*, *write_valid_only=False*, *strict=False*, ***kwargs*)

download_tasks(*task_ids*, *by_filename=False*, *validate=None*, *log=None*, ***kwargs*)

Class Inheritance Diagram



2.4.7 uccaapp.upload_conllu_passages Module

Functions

from_text(text[, passage_id, tokenized, ...])	Converts from tokenized strings to a Passage object.
glob(pathname, *[, recursive])	Return a list of paths matching a pathname pattern.
main(**kwargs)	
to_json(passage, *args[, return_dict, ...])	Convert a Passage object to text (or dict) in UCCA-App JSON :param passage: the Passage object to convert :param return_dict: whether to return dict rather than list of lines :param tok_task: either None (to do tokenization too), or a completed tokenization task dict with token IDs, or True, to indicate that the function should do only tokenization and not annotation :param all_categories: list of category dicts so that IDs can be added, if available - otherwise names are used :param skip_category_mapping: if False, translate edge tag abbreviations to category names; if True, don't :return: list of lines in JSON format if return_dict=False, or task dict if True

main

uccaapp.upload_conllu_passages.**main**(***kwargs*)

Classes

ConlluPassageUploader(user_id, ...)	
JSONDecodeError(msg, doc, pos)	Subclass of ValueError with the following additional properties:
ServerAccessor(server_address, email, password)	

ConlluPassageUploader

class uccaapp.upload_conllu_passages.**ConlluPassageUploader**(*user_id*, *annotation_user_id*, *source_id*, *project_id*, ***kwargs*)

Bases: *uccaapp.api.ServerAccessor*

Methods Summary

add_arguments(argparser)	
upload_passage(external_id, tokens)	
upload_passages(filenames, **kwargs)	

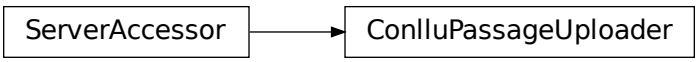
Methods Documentation

static add_arguments(*argparser*)

upload_passage(*external_id*, *tokens*)

upload_passages(*filenames*, ***kwargs*)

Class Inheritance Diagram



2.4.8 uccaapp.upload_streussel_passages Module

Functions

from_text(text[, passage_id, tokenized, ...])	Converts from tokenized strings to a Passage object.
main(**kwargs)	

Continued on next page

Table 111 – continued from previous page

to_json(passage, *args[, return_dict, ...])	Convert a Passage object to text (or dict) in UCCA-App JSON :param passage: the Passage object to convert :param return_dict: whether to return dict rather than list of lines :param tok_task: either None (to do tokenization too), or a completed tokenization task dict with token IDs, or True, to indicate that the function should do only tokenization and not annotation :param all_categories: list of category dicts so that IDs can be added, if available - otherwise names are used :param skip_category_mapping: if False, translate edge tag abbreviations to category names; if True, don't :return: list of lines in JSON format if return_dict=False, or task dict if True

main

uccaapp.upload_streussel_passages.**main**(***kwargs*)

Classes

ServerAccessor(server_address, email, password)
StreusselPassageUploader(user_id, source_id, ...)

StreusselPassageUploader

class uccaapp.upload_streussel_passages.**StreusselPassageUploader**(*user_id*, *source_id*, *project_id*, ***kwargs*)

Bases: *uccaapp.api.ServerAccessor*

Methods Summary

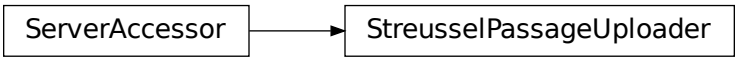
add_arguments(argparser)
upload_streussel_passage_file(filenames[, log])

Methods Documentation

static add_arguments(*argparser*)

upload_streussel_passage_file(*filenames*, *log=None*, ***kwargs*)

Class Inheritance Diagram



2.4.9 uccaapp.upload_task Module

Functions

get_passages_with_progress_bar(filename_patterns)	
main(**kwargs)	
to_json(passage, *args[, return_dict, ...])	Convert a Passage object to text (or dict) in UCCA-App JSON :param passage: the Passage object to convert :param return_dict: whether to return dict rather than list of lines :param tok_task: either None (to do tokenization too), or a completed tokenization task dict with token IDs, or True, to indicate that the function should do only tokenization and not annotation :param all_categories: list of category dicts so that IDs can be added, if available - otherwise names are used :param skip_category_mapping: if False, translate edge tag abbreviations to category names; if True, don't :return: list of lines in JSON format if return_dict=False, or task dict if True
to_text(passage[, sentences, lang])	Converts from a Passage object to tokenized strings.

main

uccaapp.upload_task.**main**(***kwargs*)

Classes

HTTPError(*args, **kwargs)	An HTTP error occurred.
JSONDecodeError(msg, doc, pos)	Subclass of ValueError with the following additional properties:
ServerAccessor(server_address, email, password)	
TaskUploader(user_id, source_id, project_id, ...)	

TaskUploader

class uccaapp.upload_task.**TaskUploader**(*user_id*, *source_id*, *project_id*, ***kwargs*)

Bases: *uccaapp.api.ServerAccessor*

Methods Summary

add_arguments(argparser)	
upload_task(passage[, log, submit, ids])	
upload_tasks(filenames[, log, submit, ...])	

Methods Documentation

static add_arguments (*argparser*)

upload_task (*passage*, *log=None*, *submit=True*, *ids=None*)

upload_tasks (*filenames*, *log=None*, *submit=True*, *existing_ids=None*, ***kwargs*)

Class Inheritance Diagram

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

s

u

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X